
Object-Oriented Numerical Methods via C++

S. D. Rajan
Arizona State University

Object-Oriented Numerical Methods

This book including computer programs is a copyrighted document. **It is against the law to copy copyrighted material on any medium except as specifically allowed in a license agreement.** No part of this book including computer programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or information storage or retrieval systems, without the express written permission of the author.

First Edition

©2002-15, S. D. Rajan

Civil, Environmental and Sustainable Engineering Program
School of Sustainable Engineering and the Built Environment

Arizona State University

Tempe, AZ 85287-5306

Phone 602.965.1712 • Fax 602.965.0557

e-mail s.rajan@asu.edu

Contents

Chapter 1	Introduction	
1.1	What is a Computer?	1-3
1.2	What is a Computer Program?	1-4
1.3	Programming in C++	1-6
1.4	What is Numerical Analysis?	1-7
1.5	What are Objects?	1-11
1.6	Why Object-Oriented Numerical Analysis?	1-11
1.7	Tips and Aids	1-12
Chapter 2	Programming in C++	
2.1	Introduction	2-3
2.2	Variables and Data Types	2-6
2.3	Expressions	2-8
2.4	Assignment Statements	2-11
2.5	Simple Input and Output	2-12
2.6	Vector Variables	2-17
2.7	Troubleshooting	2-19
2.8	The namespace Concept	2-27
Chapter 3	Control Structures	
3.1	Selection	3-3
3.2	Repetition	3-9
3.3	Other Control Statements	3-14
3.4	Tying Loose Ends	3-30
Chapter 4	Modular Program Development	
4.1	Functions	4-2
4.2	More About Functions	4-11
4.3	Scope of Variables	4-19
4.4	Developing Modular Programs	4-23
4.5	Function Templates	4-35
4.6	Case Study: A 4-Function Calculator	4-38
4.7	Exception Handling	4-45
Chapter 5	Numerical Analysis: Introduction	
5.1	Approximations and Errors	5-2
5.2	Series Expansions	5-17
Chapter 6	Root Finding, Differentiation and Integration	
6.1	Roots of Equations	6-2
6.2	Numerical Differentiation	6-6
6.3	Numerical Integration	6-12
6.4	User-Defined Functions	6-23
Chapter 7	Classes: Objects 101	
7.1	A Detour – Why OOP?	7-2
7.2	Components of a Class	7-7
7.3	Developing and Using Classes	7-19
7.4	Storage with std::vector class	7-19
7.5	String Manipulation with string class	7-26

7.5 What is struct?	7-33
7.6 Object-Oriented Solution	
Chapter 8 Pointers	
8.1 Memory Management	8-2
8.2 Pointers	8-5
8.3 Dynamic Memory Allocation	8-8
8.4 Case Study: A Poor Man's Vector Class	8-13
Chapter 9 Classes: Objects 202	
9.1 Operator Overloading	9-2
9.2 More about classes	9-12
9.3 Template Classes	9-16
9.4 Arrays	9-18
9.5 Exception Handling	9-31
9.6 Command Line Arguments	9-33
Chapter 10 Matrix Algebra	
10.1 Fundamentals of Matrix Algebra	10-2
10.2 Linear Algebraic Equations	10-8
10.3 Case Study: A Matrix Toolbox	10-34
Chapter 11 Regression Analysis (TBC)	
11.2 Interpolation and Polynomial Approximation	
11.3 Data Fitting via Least Squares Problem	
Chapter 12 File Handling	
12.1 File Streams	12-2
12.2 File Input and Output	12-2
12.3 Advanced Usage	12-9
Chapter 13 Classes: Objects 303	
13.1 Software Engineering	13-2
13.2 Inheritance	13-12
13.2 Polymorphism	13-21
13.3 Function Pointers and Functors	13-35
Chapter 14 Ordinary Differential Equations (TBC)	
14.1 Ordinary Differential Equations	
14.2 Case Study: A One-Dimensional ODE Toolbox	
Chapter 15 Partial Differential Equations	
15.1 Background	15-2
15.2 The Element Concept	15-3
15.3 One-Dimensional Boundary Value Problem	15-24
15.4 Solid Mechanics	15-27
15.5 Heat Transfer	15-31
15.6 Higher Order Elements	15-38
15.7 Mesh Refinement and Convergence	15-44
Chapter 16 Eigensystems	
16.1 Eigenproblems	16-

16.2 Properties of Eigensystems	16-
16.3 Vector Iteration Methods	16-
16.4 Transformation Methods	16-
16.5 Subspace Iteration	16-
16.6 Lanczos Method	16-
16.7 One-Dimensional Eigenproblem	16-
16.8 Case Study: An Eigensolutions Toolbox	16-
Chapter 17 Numerical Optimization	
17.1 Numerical Optimization	17-
17.2 Types of Mathematical Programming Problems	17-
17.3 Non-Linear Programming (NLP) Problem	17-
17.4 Genetic Algorithm	17-
17.5 Design Examples	17-
Chapter 18 Computer Graphics	
18.1 Introduction	18-
18.2 Simple Operations	18-
18.3 An X-Y Graphing Program	18-
18.4 Transformations and Projections	18-
18.5 Three-Dimensional Graphics	18-
18.6 Case Study: 3D Wireframe Viewer	18-

Appendix A Using Microsoft Visual C++**Appendix B C++ Standards Related Material****Appendix C Standard Template Library (STL)**

Prologue

Numerical methods constitute an important part of all scientific or engineering curricula. However, by themselves, numerical methods have limited applicability if they are not implemented as computer programs. Engineering education has realized the important role that numerical methods play in engineering analysis and design. With increasing sophistication of the analysis methods and tools, the enormous amount of data to be handled, and the need for robust, fast, easy-to-use computer programs, one cannot look at numerical analysis and computer programming as two different entities. The proposed text aims to bridge that gap.

The focus of the proposed text is three-fold.

- Understanding and practicing numerical techniques commonly encountered in scientific and engineering applications.
- Understanding the role of objects in describing and managing scientific and engineering data.
- Using C++ to develop engineering applications by linking data management with numerical methods.

The book is aimed at the junior level (3rd year in a 4-year curriculum) course in a typical engineering curriculum, or as an introductory graduate course. The material in the text is largely introductory, with some advanced topics that can be used at the discretion of the instructor. Knowledge of computer fundamentals, calculus, differential equations, and matrix algebra is required. The know how in using numerical analysis packages such as Excel[®], Matlab[®], Matcad[®], Maple[®], Mathematica[®] etc. will be a definite plus. Prior experience with writing computer programs is not assumed.

Why C++?

There are several choices one could make in selecting a programming language – FORTRAN, Basic, C, Java and C++. There are several reasons for selecting C++ as the vehicle for implementing the numerical methods. C++ is a mature language with a ready availability of mature **Integrated Development Environment (IDE)** on a variety of computer platforms. Students find these environments intuitive and useful in writing and debugging programs. The student versions of these IDEs are relatively inexpensive.

C++ supports the use of objects and all the advantages that objects afford. It is easy to write engineering applications – accuracy, speed and problem size are non-issues. Programs written in ANSI C++ can be readily extended by the addition of (system-dependent!) graphical-user interfaces (GUIs) including computer graphics. Lastly, there are hundreds of books on programming with C++ and perhaps, millions of man-hours of experience in programming with C++. This continuously evolving language has provided and will continue to provide the bulk of the engineering computer programs in the future.

S. D. Rajan
Tempe, Arizona

Introduction

“The world hates change, yet it is the only thing that has brought progress.” Charles F. Kettering

“The individual's whole experience is built upon the plan of his language.” Henri Delacroix.

“Men take only their needs into consideration never their abilities.” Napoleon Bonaparte

“Ability is of little account without opportunity.” Napoleon Bonaparte

“It is all one to me if a man comes from Sing Sing or Harvard. We hire a man, not his history.”

Henry Ford

The steam engine is said to have fueled the Industrial Revolution. In a similar vein, the microprocessor has fueled the Information Age and affected every single facet of humanity from health to education to work and play. Scientists and engineers have played a pivotal role in fueling this revolution. Developments in computer languages as well as the development of numerical analysis techniques have made it possible to create computer systems that can perform amazing tasks – allow two people thousands of miles away to communicate with each other, fly a spacecraft from the earth to Mars, help in the design and manufacture of artificial limbs, create virtual environments for the development and testing of aircrafts, automobiles, etc.

Learning a new human language can be a daunting task. But as linguists would tell you the key to learning a new language is to read, write, listen, and speak as much as possible. Learning a language involves knowing the “alphabet”, sentence construction, the grammar, ability to read, write and speak. What does one mean when one says, “I am fluent in Spanish.”? Does that indicate a fluency in reading, or writing, or speaking, or all? What does fluency mean? Are there different grades of fluency? There are no definitive answers to these questions. While the situation with computer languages is similar, there are subtle differences. Language standards developed by American National Standards Institution (ANSI) and International Standards Organization (ISO) have strongly discouraged the proliferation of language dialects. On paper, a program written using the standards should compile and execute on any hardware-software platform. These standards evolve over time and are agreed on by the Standards Committee unlike human languages that have their own evolutionary scheme. The most important

difference is that one can make mistakes in “writing and speaking” programs and learn to correct them anonymously without peer comments or stranger criticism!

Tens (if not hundreds) of computer languages have been developed and used by programmers throughout the world over the last several decades. Some of these include BASIC, FORTRAN (FORmula TRANslator), COBOL (COmmon Business Oriented Language), Lisp, Ada, Pascal, C, Smalltalk, Java and so on. The C++ language is an extension of C. Bjarne Stroustrup developed this language in the early eighties at AT&T Bell Laboratories. It would be incorrect to refer to C as a subset of C++ or C++ as a superset of C. C++ has features that make it a programming language of choice for business, scientific and engineering applications.

We hope this book will open your minds (and the doors) to making this world a better place to live in.

Objectives of this book

- To understand why C++ is a suitable computer language for the development of software for scientific and engineering analysis and design.
- To study and understand the different numerical analysis techniques which are useful for engineering analysis and design.
- To study and understand the basic concepts associated with software development.
- To understand the basics of object-oriented (OO) programming.
- To apply OO techniques in the development of software-based numerical solution techniques.

The constructs of the C++ language are slowly introduced throughout the text. The basics of the language are introduced in Chapter 2. More useful ideas and constructs are discussed in Chapters 3 and 4. This background is enough to start a serious study of numerical analysis techniques. In Parts I and II, we start with basic ideas such as approximations and series expansions in Chapter 5 followed by roots of equations and numerical integration and differentiation in Chapter 6. In Chapter 7 we see a gentle introduction to object-oriented (OO) ideas. The applications of OO ideas especially with regards to development of scientific and engineering software development are shown in Chapters 8 and 9. Having introduced some of the basic building blocks of numerical analysis – vectors and matrices, we move on to Part III where we see a number of matrix operations including solutions to linear algebraic equations. We follow this with associated numerical analysis ideas – interpolation, polynomial approximation and curve fitting.

We see more advanced ideas and topics in the second half of the book. In Chapter 12, file handling constructs are discussed. We follow this in Chapter 13 with more advanced ideas dealing with classes and objects such as inheritance and polymorphism. In Chapters 14 through 17, we see important numerical analysis ideas dealing with ordinary differential equations, partial differential equations, eigenproblems and numerical optimization. Finally in Chapter 18, we see an introduction to computer graphics.

1.1 What is a Computer?

A computer is a sophisticated tool. It is typically made up of hardware, firmware and software. Hardware (see Fig. 1.1) includes components such as a computer case containing the motherboard, central processing unit (CPU), random access memory (RAM), CD-ROM drive, floppy drive, magnetic disk (hard disk), video card, sound card, network interface card (NIC) etc., along with output devices such as monitor, printer etc. and input devices such as keyboard, mouse etc. Software is defined as programs, routines, and symbolic languages that control the functioning of the hardware and direct its operation. The operating system (OS) is an example of software as are word processors, drawing programs, video-editing programs, CAD programs, weather forecasting programs, web browsers etc. Firmware includes coded instructions that are stored permanently in read-only memory (ROM). For example, when the computer is powered on, instructions from the firmware are used to load the OS from the disk and pass control to it.

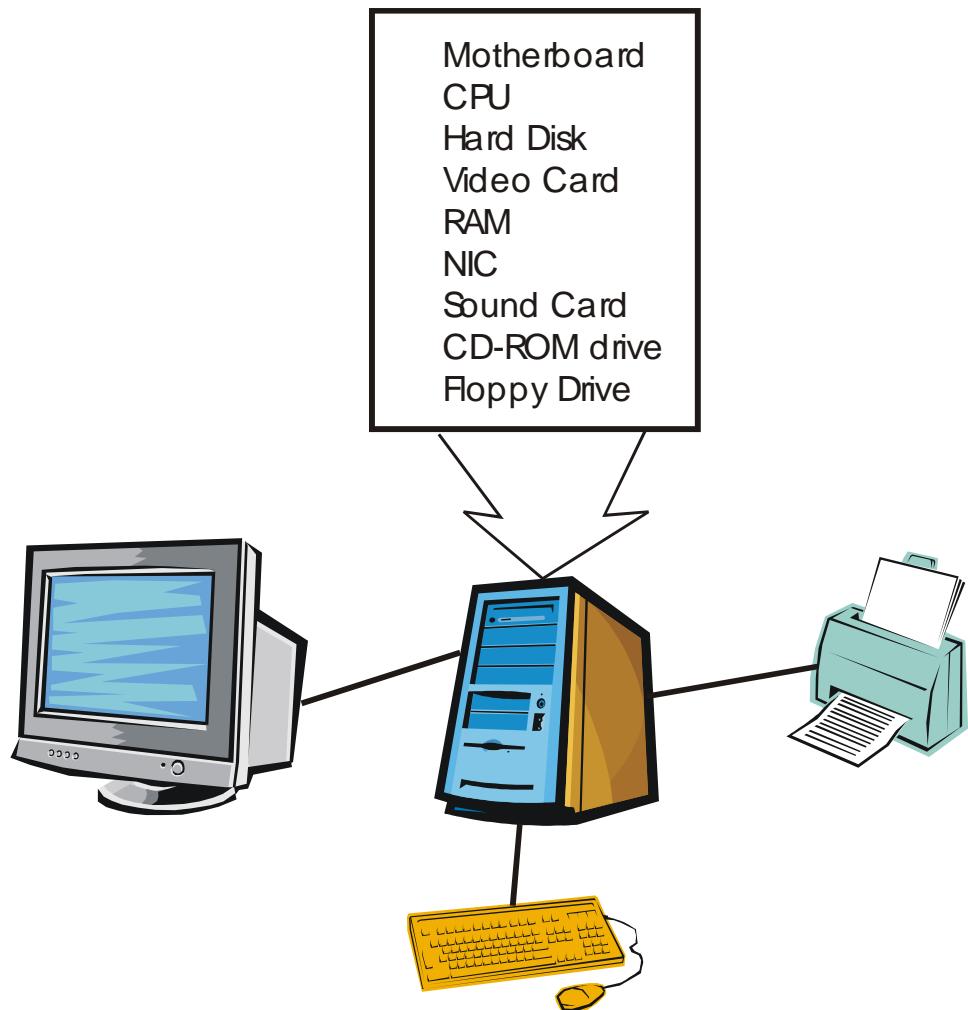


Fig. 1.1 Components of a computer system

Hardware advances are taking place at a breathtaking speed whereas advances in software are at a relatively much slower rate. In spite of these advances, increases in human productivity at the workplace seem to be nebulous. Some of the problems seem to be because of a lack of understanding of what technology is and how humans need to interact with technology. Successful deployment of computer-based tools is not just a function of computer hardware or software but involves much more as we hope to see in this text.

1.2 What is a Computer Program?

A computer program contains low-level instructions (or machine instructions) that enable information (instructions and data) flow between different components of a computer. How do we develop a computer program? The process typically starts with a need for a computer program. Based on the needs one would develop a set of specifications that detail the capabilities of the computer program, on what computer platforms it needs to run, how it would interact with the users, the type and nature of input that would drive the program and the type and nature of the output that the program would generate. The computer programmer or software engineer would then take the capabilities specifications and translate it into implementation specifications. A short answer is that the programmer would develop the source code, compile the source code and create the object code, link the object code to create the executable image (computer program) and then execute the computer program on a specific hardware-software computer system. Let us now see what are meant by these terms.

Source Code: Source code is made up of the statements in a computer programming language. Here are the source statements of a computer program in FORTRAN that converts weight from pounds to Newtons.

```
C      PROGRAM CONVERT
C
C      CONVERTS POUNDS TO NEWTONS
C
C      REAL POUNDS, NEWTONS
C
C --- GET THE USER INPUT IN POUNDS
      WRITE (*,100)
100 FORMAT (1X, 'INPUT WEIGHT IN POUNDS: ')
      READ (*,*) POUNDS
C
C --- CONVERT TO NEWTONS
      NEWTONS = 4.448*POUNDS
C
C --- DISPLAY THE CONVERSION
      WRITE (*,101) POUNDS, NEWTONS
101 FORMAT (1X, 1PE15.8, ' POUNDS IS EQUAL TO ', 1PE15.8,
      1           ' NEWTONS')
C
C --- ALL DONE.
      STOP
      END
```

Typically, one would create the source code using an editor. An editor is a computer program that allows the creation of and subsequent editing of the contents of a text file. Microsoft Word[©] or Notepad are examples of an editor. Programmers use a custom made environment called an Integrated Development Environment (IDE) to develop, write, edit, debug and execute computer programs. The source statements are stored in one or more files and they have a special file extension. C/C++ source files have file extensions as .cpp, .c, .h etc.

Compile: Once the source statements are ready, they need to be compiled. A compiler (another computer program!) takes the source statements (in one or more files) and creates object files. The compiler issues warnings and error messages if the source statements do not follow the C++ syntax. One can look at object files as intermediate files that by themselves cannot be used but are needed to create the executable image. Object files have file extensions as .obj, .o etc.

Link (or Build): Once all the source statements are compiled and the object files are created, the linker (another computer program!) is used to “tie” the object files to other C++-enabled components (or libraries) so as to produce the executable image. Executable files have file extensions as .exe etc. though on Unix systems, by default, the linker created executable image is called a .out. If the linker is not able to find all the components, it will issue error messages and the executable image is usually not created.

Execute: Once the executable image or the program is created, it can be executed on the hardware-software platform for which it was developed. Programs may not execute correctly because of either logical errors or run-time errors. If a program runs from start to finish without any errors but does not produce the correct output is said to have logical errors. On the other hand, if the program “crashes” during execution then it has encountered a run-time error. Examples include illegal operation (divide by zero, overflow, underflow, etc.), illegal memory access (access violation) etc.

Debug: Programs rarely work correctly the first time. Finding and correcting both logical errors and run-time errors are challenging. However, there are systematic approaches and debugging tools that can be used to find these errors in programs small and large.

Writing excellent computer programs is an art whereas the skills necessary for writing good computer programs can be learnt through good programming habits (much as learning good scientific or engineering practices). There are three distinct components that we must deal with in our quest to program effectively. First, we have the language itself. Second, is the environment in which the computer programs are developed, written, debugged and executed. A good IDE certainly helps. Learning to effectively use IDE tools – editor, debugger etc. can be a life saver. Last, we have the programmer with his or her thought processes, ability to visualize, plan and execute, idiosyncrasies, troubleshooting capabilities, and hopefully, a whole lot of patience.

1.3 Programming in C++

The CEO of Frame and Truss

*Told his employees "Thou shan't cuss"
 If you so hate FORTRAN
 Then join the clan
 That writes its programs in C++*

As we will repeatedly see in this book, skills can be mastered through practice and hard work. Note that there is nothing more satisfying than a completed, running program however little or much it may do!

An interesting site¹ that can be used as a starting point to have answers to questions about C++ is

<http://www.parashift.com/c++-faq-lite/>.

I will use four (philosophical) questions and answers from that site here.

Question: Is C++ a practical language?

Answer: Yes. C++ is a practical tool. It's not perfect, but it's useful.

In the world of industrial software, C++ is viewed as a solid, mature, mainstream tool. It has widespread industry support which makes it "good" from an overall business perspective.

Question: Is C++ a perfect language?

Answer: Nope. C++ wasn't designed to demonstrate what a perfect OO language looks like. It was designed to be a practical tool for solving real world problems. It has a few warts, but the only place where it's appropriate to keep fiddling with something until it's perfect is in a pure academic setting. That wasn't C++'s goal.

Question: What is the big deal with OO?

Answer: Object-oriented techniques are the best way we know of to develop large, complex software applications and systems.

OO hype: the software industry is "failing" to meet demands for large, complex software systems. But this "failure" is actually due to our successes: our successes have propelled users to ask for more. Unfortunately we created a market hunger that the "structured" analysis, design and programming techniques couldn't satisfy. This required us to create a better paradigm.

C++ is an OO programming language. C++ can also be used as a traditional programming language (as "as a better C"). However if you use it "as a better C," don't expect to get the benefits of object-oriented programming.

Question: Is C++ better than Ada? (or Visual Basic, C, FORTRAN, Pascal, Smalltalk, or any other language?)

¹ From Marshall Cline's C++ FAQ Lite document.

Answer: This question generates much much more heat than light. Please read the following before posting some variant of this question.

In 99% of the cases, programming language selection is dominated by business considerations, not by technical considerations. Things that really end up mattering are things like availability of a programming environment for the development machine, availability of runtime environment(s) for the deployment machine(s), licensing/legal issues of the runtime and/or development environments, availability of trained developers, availability of consulting services, and corporate culture/politics. These business considerations generally play a much greater role than compile time performance, runtime performance, static *vs.* dynamic typing, static *vs.* dynamic binding, etc.

Anyone who argues in favor of one language over another in a purely technical manner (i.e., who ignores the dominant business issues) exposes themself as a techie weenie, and deserves not to be heard.

It should be noted that a programming language is a vehicle; the programmer is the driver and must chart the course.

1.4 What is Numerical Analysis?

There are hundreds of engineering and scientific problems that can be solved analytically. Consider the simple example of finding the roots of a quadratic polynomial of the form $f(x) = ax^2 + bx + c$. The analytical solution to the real roots is $x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ provided $\sqrt{b^2 - 4ac} \geq 0$. When this

problem is presented with numerical values for the constants a, b and c , one can find the roots by using a pencil and paper, or a calculator. When the problems become more complex, analytical solutions are either too difficult or time consuming to obtain, or it is simply not possible to obtain. In cases such as these, it may be possible to obtain an approximate, numerical solution.

There are at least three components to obtaining the numerical solution. First, there must exist a procedure, method, or technique that establishes the link between the input to the problem and the expected output from the solution methodology. Second, it is necessary to translate the solution methodology to a finite set of distinct steps or algorithm. Last, we must be able to take the algorithm and convert it to a computer program. Let's look at the second and third components in the next two sections. The rest of the book is about the first component!

Algorithm

Solutions to problems usually follow a specific path – examination of the problem statement, identification of the problem parameters by differentiating between the input and the output parameters, recognizing what theoretical details (methods, techniques etc.) are applicable, and finally, development of the algorithm that bridges the gap between the input and the output parameters.

For the problems discussed in this text we will define the solution process in terms of algorithms. An algorithm is a sequence of detailed steps that is general enough to be applicable for most situations in solving a problem. These steps involve the input variable(s) to the procedure, and lead to the determination of the output variable(s).

Steps in a typical algorithm involve input, output, assignment, and control structures.

Input and Output

Every algorithm involves either input or output or, more often than not, both. For example, the general procedure for the analysis of beam deflections using the solution of an ordinary differential equation uses the beam cross-sectional and material properties, the different span lengths, the loading on the different spans, and the manner in which the beam is supported as input to generate the output – the rotations, displacements, and the internal forces along the beam.

Assignment

An assignment in the form of an equation or expression is made up of one or more of algebraic operators, variables, constants, and mathematical functions. Examples of algebraic operations include addition, subtraction, multiplication, division etc. Examples of mathematical functions and operators include \sin , $| |$, $\sqrt{ }$ etc. We will be introduced to specific C++ examples of assignments in Chapter 2.

Control structures

While an algorithm may have several steps, the computations are executed in a specific order that may involve only a few steps, or certain steps may be executed repeatedly. Control structures help the programmer in sequencing the execution of the steps in an algorithm. Research in program development has shown that control structures can be divided into three types – the *sequence structure*, the *selection structure*, and the *repetition structure*. Sequence structure means that statements are executed in order – this is the manner in which an algorithm is developed in terms of ordered steps. As the name suggests, the selection structure involves execution or skipping of specific steps in an algorithm. Finally the repetition structure involves repeated execution of a sequence of steps. We will be introduced to specific C++ examples of control structures in Chapter 3.

We will illustrate these ideas through an example.

1.4.1 A Sample Algorithm

Consider the problem of finding a root, \hat{x} of a nonlinear equation given as $f(x) = 0$. A well-known solution technique is the Newton-Raphson Method that we will see in detail in Chapter 6. The basic idea is to start with an initial guess, x^0 for the root. A better estimate x^{k+1} for the root is generated as

$$x^{k+1} = x^k - \frac{f(x^k)}{f'(x^k)} \quad k = 0, 1, 2, \dots \quad (1.4.1.1)$$

The iterative process of finding a better estimate continues until an appropriate termination criterion is reached. For example, one could establish the maximum number of iterations, k_{\max} , or one could compare the change in the estimate against a predefined tolerance, ε , as $|x^{k+1} - x^k| < \varepsilon$, or compare the change in the function value against a predefined tolerance, δ , as $|f(x^{k+1}) - f(x^k)| < \delta$, or even $|f(x^{k+1})| < \delta$.

Before a computer program is written, one must translate the theory and process discussed above into an algorithm. A good algorithm is a detailed set of instructions that a computer programmer can use to translate into appropriate computer statements.

Algorithm for Newton-Raphson Method

- (1) Input: Establish the values for k_{\max} , ε , δ , γ (see Step 3) and the initial guess x^0 .
- (2) Set $k = 0$.
- (3) Compute $f(x^k)$, $f'(x^k)$.
- (4) If $|f'(x^k)| < \gamma$, note that the solution did not converge and go to Step 8.
- (5) Otherwise, use Eqn. (1.4.1.1) to find the next estimate x^{k+1} .
- (6) Check the termination criteria. If one or more criterion is satisfied, then go to Step 8.
- (7) Otherwise, increment $k = k + 1$. Go back to Step 3.
- (8) Output: Message as to whether the solution converged or not. If the solution converged, then the values of \hat{x} , $f(\hat{x})$ and k are useful output values.

Steps (2) through (7) involve repetition. Selection is shown in Step (4) and Step (6). Note Step (4). The second term in Eqn. (1.4.1.1) is singular if $f'(x^k) = 0$. However, the term is numerically singular (leading to an exception - numerical overflow), if $f'(x^k)$ is numerically small. An exception is a kind of error that may prevent a program from progressing further and from producing correct output. Assignment takes place in Step (5).

1.4.2 Implementing Algorithms

The next step in the overall process is to translate the algorithm into a program or program component. However, there are still unresolved issues such as program architecture, data structures, program input and output that the programmer must have answers to before a computer program is actually developed.

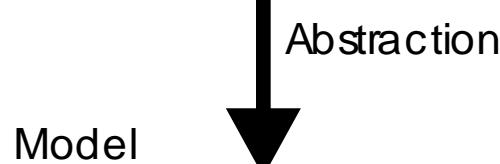
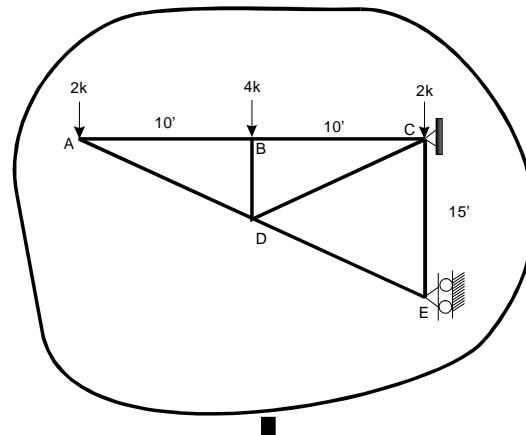
In some sense, we are jumping ahead of ourselves. The thought process starts with statements that describe the problem and your abstract view that describes the model for the problem. For example, if

the main objective is to develop computer programs to carry out the structural analysis of structural systems such as planar trusses. A typical abstraction may look as shown in Fig. 1.4.2.1.

The model is an abstraction. For example, what does the statement “Define a truss” mean? How do we go from input data to generation of output? How do we define and store the data associated with a truss? How general is our model? What are its limitations? What steps should be followed to turn this abstract model into a functional program? What is the role of algorithms in this model?

We will see some of these issues discussed throughout the text.

Problem Description



Model

```

...
Define a truss
...
Read the input data.
...
Analyze.
...
Generate the output.
...

```

Fig. 1.4.2.1 Creating an abstract model

1.5 What are Objects?

Objects are routinely identified by human beings. They are given a name, defined as having properties (what is it?) and capabilities to do something (what does it do?). Computer scientists term these characteristics as *attributes* and *behavior*. Let us look at a few examples.

A *person* can be thought of as an object. The attributes for a person include the person's name, gender, height, weight, age etc. Some of the questions that we should be able to ask about the person include "Is this person eligible to vote?" (determined based on the age), "Will this person fit inside a NASA capsule?" (determined based on the height), etc.

Time is an object. It is defined in terms of hour, minute and second. These are its attributes. Some of the questions that we should be able to ask about time include the following - "Is it past breakfast time?", "How much time has elapsed since 9 a.m.?", "How do we print time in different formats?" etc.

Construction material from a viewpoint of a structural engineer is an object. The material is identified by a name, e.g. A36 Steel, Aluminum 2014-T6 etc. In addition, it has structural properties of interest to us – modulus of elasticity, mass density, yield stress etc. These are the possible attributes for a material object. We would like to know "Is the material elastic with a normal strain value of 0.02?", "How much does a cubic meter of the material weigh?", etc.

In a computer program, the objects are defined in terms of classes. *An instance of a class is an object* in a manner similar to

```
int x;
```

where **x** (object) is an instance of type integer (class). One can think of classes as user-defined data type. We should be able to set as well as obtain the values for some or all the attributes that define an object. We would also like to have answers to the behavioral aspects of the class. Such features are available through the class interface. We will see more about classes in Chapter 7.

1.6 Why Object-Oriented Numerical Analysis?

Large software developers will tell you that OO technology has revolutionized software engineering. It is changing the way in which software is developed, reused, and enhanced. Here are some excerpts about the benefits of OO technology [Lee and Tepfenhart, 2001].

- The proficiency of higher-level OO model should provide the software designer with real-world, programmable components, thereby reducing software development costs.
- Its capability to share and reuse code with OO techniques will reduce time to develop an application.

- Its capability to localize and minimize the effects of modifications through programming abstraction mechanisms will allow for faster enhancement development and will provide more reliable and more robust software.
- Its capability to manage complexity allows developers to address more difficult applications.

Object technologies lead to reuse, and reuse of program components leads to faster software development and higher-quality programs. Object-oriented software is easier to maintain because its structure is inherently decoupled. This leads to fewer side effects when changes have to be made and less frustration for the software engineer and the customer. In addition, object-oriented systems are easier to adapt and easier to scale – large systems can be created by assembling reusable subsystems [Pressman, 2001].

1.7 Tips and Aids

Finally a few words about getting the most from this book and from a course on numerical analysis and computer programming. The presentation style used in this text has two emphases. First, the relevant syntax, theory and/or assumptions are presented that describe the C++ language, a numerical analysis technique or method. Second, the syntax or theory is followed by several examples that serve to illustrate different traits of the C++ language syntax, problem-solving abilities of the method, technique or theorem. Every example has been chosen with care. You are encouraged to look at each and every example in detail. Ideas are explored and comments made so that we are able to appreciate the different nuances of the theory and its applications. Finally, every section is followed by several problems that you are encouraged to solve. Some have answers that can be used to check your solution.

I have accumulated over the years, some perspective on effective teaching and learning.

- Come prepared to class. Spend about 15-30 minutes reading the material for that day's lecture. Read it more than once if necessary even if you do not understand most of the material. Familiarity with the “language of the lecture” – terminology, figures, equations etc. is a major advantage that you will carry with you to the lecture.
- At the end of the day (not the end of the week), carefully review the lecture material – language syntax, algorithm, derivations, solved examples etc. Look at this task as changing the oil in an automobile – either you do it regularly or you take the car to a mechanic for a major repair job later. Close the lecture notes and resolve the solved examples from start to finish. Go back to the lecture notes if you are unsuccessful. Repeat the exercise until you successfully solve the problem. Too often we are tempted to say “I understand the material but ...”. There is simply no substitute for solving a problem from **start to finish**.
- Practice, practice and practice. You will have a better understanding of the material by solving problems.

- Don't be afraid of asking questions. A correctly posed question can be a huge life and time saver. Your ability to ask the correctly posed question is directly proportional to the amount of "homework" you do.
- Use every conceivable resource – the instructor's knowledge and office hours, the TA's enthusiasm, knowledge and office hours, the library, material available on the internet etc. This is perhaps the only time in your life when you will have the time, energy, motivation, atmosphere and resources available simultaneously to meet your needs.
- Study in small groups associating with other students who are as motivated, diligent and capable, if not better, than you are. Otherwise this is likely to be a waste of time. Put this study session into your weekly schedule.
- A student should be ethical before he or she becomes a practicing scientist or engineer. When it is not clear as to what is ethical, consult someone knowledgeable.
- Successful exploitation of numerical analysis and computer programming requires understanding of the material, patience, practice and application. One of the most attractive aspects of numerical solutions is that we can develop the entire solution, apply and debug the solution in a virtual environment. Do not hesitate to experiment with ideas or be innovative. Understand the problem, be systematic, and when required, don't ignore the details. The devil is in the details.

It is hoped that the material in this text encourages and helps you in being a better scientist or engineer and an even better student.

Tip: This would be the right time to install the Integrated Development Environment (IDE) that you wish to use. The Microsoft Visual Studio environment is discussed in Appendix A. Go ahead and install the IDE and familiarize yourself with the different components.

There are other (other than Microsoft) C++ compilers available for the Windows operating system. These include compilers from Intel (that is integrated into MS Visual Studio), g++ under cygwin, Bloodshed Dev-C++, Digital Mars and others. In addition, there are other IDEs available on other operating systems. Notable among them are IDEs and C++ compilers for the Linux operating system such as Intel, Portland Compiler Group, Eclipse, g++ etc.

EXERCISES

Appetizers

Problem 1.1

Research the world wide web to find answers to the following questions.

- (a) What other computer languages not mentioned in this Chapter were or are being used by programmers? Write a short paragraph on each of these languages.
- (b) What hardware advances have taken place in the last 5 years that are now available in computer systems?
- (c) What are the most commonly used operating systems? What hardware platforms do these operating systems require?

Problem 1.2

Write an algorithm for finding the maximum of a set of numbers.

Problem 1.3

Write an algorithm for finding the minimum of a set of numbers.

Problem 1.4

Consider an airplane as an object. Identify its attributes. List its behavior that one may want to incorporate in a computer program.

Problem 1.5

Are the following hardware, software, firmware or none of the above? (a) Microsoft Excel[®] (b) CPU (c) Adobe Acrobat[®] (d) Cache (e) BIOS (Basic Input/Output System) (f) Personal Device Assistant (PDA).

Main Course

Problem 1.6

Consider a point in space defined in terms of its (x, y, z) coordinates. Write an algorithm (including error detection) for each one of the following tasks.

- (a) Compute the distance between two points.

- (b) Compute a unit vector between two points.
- (c) Compute the shortest distance of a point from a straight line. Assume that the line is defined by two points.

Problem 1.7

Consider each of the following as an object. Identify their attributes. List their behavior that one may want to incorporate in a computer program. (a) Time (b) Bank account (c) Employee (d) Fuel sources.

Problem 1.8

Make a list of frequently asked questions (FAQs) on how to use your favorite IDE. Get the answers to these questions.

C++ Concepts

Problem 1.9

Write an algorithm for the tic-tac-toe game. Make the following assumptions. The game will be played exactly once. Who (user or computer) will play first will be determined by a coin toss. The person who plays first marks an **X** and the other person uses an **O**. The cells in the grid are identified by numbers 1 through 9 - the user inputs a valid number 1 through 9. You have to develop the strategies for the computer (aim being to win the game if possible).

Problem 1.10

Describe the steps that you would take to convert the ideas from *Problem 1.9* into a computer program.

Programming in C++

"Man invented language to satisfy his deep need to complain." Lily Tomlin.

"A man of great memory without learning bath a rock and a spindle and no staff to spin." George Herbert

"Real programmers can write assembly code in any language." Larry Wall.

In this chapter we will see the power of the C++ language through simple yet powerful programs. We will start by examining a short and complete program to show the different components that are found in most C++ programs. Learning to use C++ is in some ways learning a human language. You will have to learn the syntax of the language. You will have to practice on how to use the language correctly. You will have to seek the help of more experienced people when you encounter a problem. Sometime, you will have to put logic and common sense aside and accept the idiosyncrasies of the language. And finally, you will have to be very patient, organized and determined.

Use a hands-on approach to programming. Try to build and execute the program. There is simply no substitute for practice, practice, and practice in learning how to develop compute programs. It is highly recommended that you use an Integrated Development Environment (IDE) to develop, debug and refine your programs. What is an IDE? An IDE is an environment (a visual computer program) that gives a programmer a variety of tools to make the development and maintenance of programs easier to do. A typical IDE provides (a) an editor to create and edit the source code, (b) a program “make” capability – a tool that instructs the compiler and the linker on what and how to compile and link a program, and (c) an interactive debugger that would help the programmer step through and debug the program etc. Microsoft Visual Studio is an example of an IDE that provides an environment for building applications or programs in a variety of languages – Basic, C++, FORTRAN etc.

This chapter is just the beginning. However, it will provide a quick jump start and rapidly introduce several useful features. A full understanding of the features will come as we use a feature over and over again throughout the text.

C++ is an extremely powerful language. We will learn more about the language features and capabilities throughout this text. Finally, it is recommended that you go though the entire chapter, perhaps more than once, in order to get a firm grasp of the basics – creating, editing, compiling, linking, executing and debugging a program.

Objectives

- To understand the basic syntax of C++ programs.
- To understand the concepts associated with data types, variables, arithmetic expressions, assignment statements and simple input and output.
- To understand and practice writing complete C++ programs.
- To compile, link and execute programs.
- To learn the art of troubleshooting.

SUPPLEMENTAL MATERIAL

All sample programs shown in this text are available on the book web site. The programs have been compiled and executed using Microsoft Visual Studio 2008 compiler running under Windows XP/Vista/7. Unless otherwise noted, they should compile and execute on all ISO-compliant systems.

2.1 Introduction

Every language, whether human or computer based, has its syntax. We must recognize and use the syntax. Unlike human languages and their usage, the computer language syntax is unforgiving and hence must be followed correctly. If we don't, it would be impossible not only to execute computer program correctly but also it may not be possible to build a computer program. Below we present a small but complete C++ program. This program is designed to display the string **Welcome to Object-Oriented Numerical Analysis.** on the screen.

The source code to a typical computer program is made up of several lines of input contained in a text file. A text file is a file that can be viewed in an editor or viewer such as Microsoft Windows Notepad. More often than not, the file extension associated with a C++ source file is **.cpp**. For example, naming a file **main.cpp** would imply that the file contains C++ source code.

Example Program 2.1.1 A simple C++ program

In the example shown below, the contents of the actual text file (also called source code) are shown in **blue**. The **line numbers** are shown so that references to individual lines can be made later in the text. C++ keywords are in **red**.

main.cpp

```

1  /* EXAMPLE 2.1.1
2   Copyright(c) 2001-08, S. D. Rajan
3   Object-Oriented Numerical Analysis
4
5   OBJECTIVES
6   (1) Illustrate a simple but complete program.
7   (2) Show the compile, link and execute steps.
8
9  */
10
11 #include <iostream> // header file
12 using std::cout;    // standard output class
13
14 int main()
15 {
16     cout << "Welcome to Object-Oriented Numerical Analysis.\n";
17
18     return 0;
19 }
```

The example shows some program components of a long list that you will possibly find in a C++ program. The lines of input in a C++ program are free format input meaning that one can type the input starting at any location in a line. One can also break the input into several lines. There are exceptions to this rule as we will see later.

How are the statements executed? In C++ as in most other languages, the statements are executed sequentially as they appear in the program. In other words, the statement in line 1 is executed first,

followed by the execution of the statement in line 2 and so on. Sometimes, some lines are not executed because they are comment lines and sometimes the program logic requires that these statements be skipped.

Lines 1 through 9 are comment lines. These lines are ignored by the compiler. Good programming habit is to add comment lines to all programs not only to explain to ourselves what is being done in the program but also to help others who may have to use our computer programs. A comment section starts with the pair `/*` and ends with the pair `*/`. Line 10 is a blank line that has been used to improve the readability of the program as are lines 13 and 17.

A typical C++ program will not be totally self-contained. It will leverage components created by others. Those starting new with C++, may find it confusing as to what is a part of the language, what is not and how the language can be extended by the programmer to meet the programmer's requirements. C++ has a set of reserved keywords and symbols (or tokens) that are a part of C++ language. In other words, as a programmer, these keywords and symbols should be used in a specific manner – following the syntax associated with the keyword or symbol. For example, the reserved keywords used in the program are `include`, `using`, `int`, `main`, and `return`. Each of these keywords has a special syntax. For example, the `#include` has the following syntax.

```
#include filename
```

The reserved symbols used in the program are `#` ; { } " " \ / * : <>. C++ provides additional functionalities through the use of library functions etc. If we wish to use them, then we must explicitly state in our program what functionalities are being used. The cardinal rule in C++ is that declarations must precede usage. Otherwise there is ambiguity in the statements that follow. We will introduce and discuss these reserved symbols throughout the text.

Going back to the example, we need to find a way to display the string on the computer screen. The correct terminology is output the string to the screen. C++ provides several classes for input and output. The class that is used here is called `iostream`. The information about the `iostream` class is contained in the file called `iostream` (such files are called header files¹). The specific object associated with this class that is most commonly used for outputting streams of characters is called `cout` (`cout` is an `ostream` class object). Whenever `cout` is used, it must be followed by the operator `<<` that is then followed by the information to be displayed contained in a proper form. We will see more about this in Section 2.5. Since the `cout` statement is used later in the program, the `include` statement is used to tell the compiler that we wish to use the `iostream` class (line 11). In line 12, we qualify this fact further by stating that we wish to use the (standard) `cout` object in the `iostream` class. The syntax will become clearer when we look at objects in Chapter 7. The `#` sign used before the `include` keyword, must be placed in the first column (some compilers will accept `#` as the first nonblank character in the line). The declarations in lines 11 and 12 make it syntactically correct when `cout` is used in line 16. Also note the alternate form of comments on a single line. Anything that follows (to the right of) `//` on a line is treated as a comment and is ignored.

¹ There is a difference between the usage `#include <iostream>` and `#include <iostream.h>` that we will see later. A header file contains function prototypes that give the compiler information on the functions used in a program.

Every C++ program must have one and only one special function called `main`. A typical function is a program component that is usually called from other parts of the program. They are used to simplify the development of computer programs. `main` is a function! However the `main` function is not called in any program. Instead the program execution starts in the `main` function. We will study functions in greater detail in Chapter 4. The `main` function starts on line 14. The `int` keyword signifies that an integer value will be returned from the `main` (function) to the program that calls `main`. The body of `main` is contained within the symbols `{` and `}`. These symbols are known as curly braces. Hence, the three lines 16 through 18 form the body of `main`. Line 16 carries out the only task that this program is designed for. In this example, the literal contents of the character string between " " symbols are output to the screen except for `\n`. `\n` is a special formatting symbol that signifies a new line character should be sent to the screen so that the cursor rests on the next line. The generated output is shown in Fig. 2.1.1(a). Fig. 2.1.1(b) shows the program output if `\n` is not used. Finally, since we declared `main` to return an integer value, the last statement in line 18 has the statement to `return` a zero value. Also note that the `;` (semicolon) symbol is used to terminate a statement. The syntax associated with the use of a particular feature will indicate if a semicolon is required or not. As we can see from this example, the `using` directive, the output via `cout` and the `return` statements require a semicolon to terminate the statements.

Go ahead and compile, link and execute Example 2.1.1.

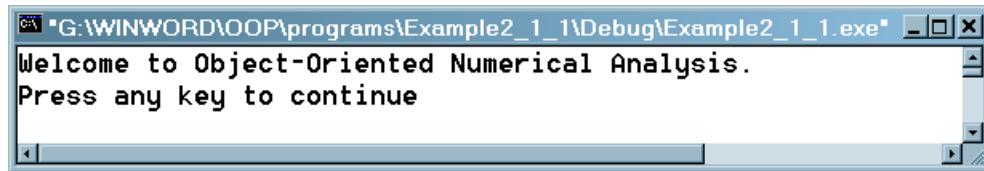


Fig. 2.1.1 (a) Output generated by Microsoft Visual Studio

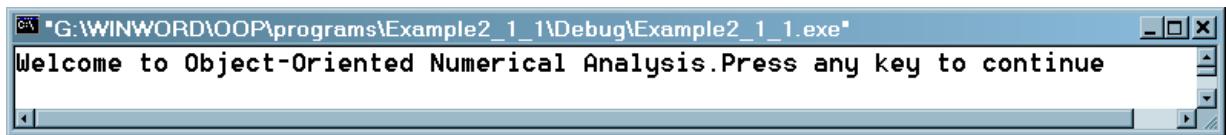


Fig. 2.1.1 (b) Output generated by Microsoft Visual Studio

Tip: C++ specifies a proper format for the usage of keywords and symbols. Keywords are case sensitive - `Cout` is not the same as `cout`. Similarly, blank spaces must be used with care - `< <` is not the same as `<<`. If something is declared, you may elect not to use it. However, it is a wasteful practice to declare variables or functions but not to use them.

Some of the more common components of a C++ program are briefly discussed below.

Header files: Header files are separate files that contain C++ statements and are typically used in other files via the `#include` statement. Header files usually have the `.h` file extension.

Variables: Variables are used to store values. The values are assigned to the variables and possibly changed over the course of execution of a program.

Expressions: Expressions are created using constants, variables, operators, functions etc. Expressions are evaluated and the result is a value.

Statements: A statement is made up of one or more of the following - variables, expressions, C++ keywords and tokens.

Input and Output: Information is acquired by the program as input from the keyboard, mouse, file, etc. and information is sent by the program to the monitor, file, printer etc.

Functions: A function is a program component that can be called from other parts of the program. We will see the syntax and functionalities of functions, and when and where functions should be used later in Chapter 4.

We will examine some of these components in the rest of this chapter.

2.2 Variables and Data Types

As was stated in the introductory chapter, most of the computer programs store and manipulate data or information. C++ provides supports for several types of data. The fundamental ones are shown below. We will learn more about bits , bytes and words in Chapter 5.

Data Type	Values (System dependent)	C++ Example
<code>short</code>	16 bits long. Integer value from -32767 to 32767	-145
<code>int</code>	32 bits long. Integer value from -2 147 483 648 to 2 147 483 647	1034
<code>long</code>	64 bits long. Integer value from -9 223 372 036 854 775 808 to 9 223 372 036 854 775 807	80967845L
<code>float</code>	32 bits long. Floating point value in the range $\pm 3.4(10^{\pm 38})$ (7 digits precision)	-0.0035f
<code>double</code>	64 bits long. Floating point value in the range $\pm 1.7(10^{\pm 308})$ (15 digits precision)	1.45
<code>bool</code>	false or true	true
<code>char</code>	1 character	g

An integer value unlike a floating point value has no fractional component. The values -345 and 12000 are examples of integer values. The values 1.34 and -0.0045 are examples of floating point values. By

default, all floating point numbers are of type `double`. Hence, `1.34` is a double constant whereas `1.34f` is a float constant. Note that an `L` at the end of an integer value signifies that the constant should be treated as a long data type. Similarly, an `f` at the end of a float number signifies that the number is a float data type. As we will see later, there are other types of data including ones that the user or programmer can define and store.

To store values that may change over the course of execution in a program, we use variables. In other words, variables can be manipulated to store values. A variable is identified with a name. A variable name can have many characters, starts with an alphabet, and typically is made up of the following characters – a through z, A through Z, 0 through 9, and `_`. No blank spaces are allowed. Examples of valid variable names are given below.

`a345`, `z_helper`, `Alpha`

The following variable names are **invalid**.

`1Alpha`, `a Temperature`, `B$65`

Note that each variable can store only one value. Such a variable is called a scalar variable. When several values need to be stored, we can use a vector variable. We will look at vector variables later in the chapter.

To understand the usage of the variables in a computer and the type of data that is stored in them, we will use the following convention. You may find that other books or software firms have different conventions. Some of the data types have not been discussed as yet and can be ignored for now.

Data Type	Variable Name Prefix	Examples
Integer scalar	<code>n?</code>	<code>nX</code> , <code>nIterations</code> , <code>nJoints</code>
Float scalar	<code>f?</code>	<code>fY</code> , <code>fTolerance</code>
Double scalar	<code>d?</code>	<code>dArea</code> , <code>dP123</code>
Boolean scalar	<code>b?</code>	<code>bDone</code> , <code>bConverged</code>
Integer vector	<code>nV?</code>	<code>nVScores</code> , <code>nVSSN</code>
Integer matrix	<code>nM?</code>	<code>nMVertices</code> , <code>nMShapes</code>
Float vector	<code>fV?</code>	<code>fVRHS</code> , <code>fVForces</code>
Float matrix	<code>fM?</code>	<code>fMCoef</code> , <code>fMElementForces</code>

Double vector	dV?	dVRHS, dVGPA
Character	c?	cGrade
String	sz?	szNames, szStates

Here are a few examples illustrating how we can declare variables for different data types in a computer program.

```
int nV;           // one integer variable
int nA=1, nB=3, nC=5; // three integer variables with initialization
int nA(1), nB(3), nC(5); // three integer variables with initialization
double dPrecision; // double precision variable
float fX, fY, fZ; // float variables
bool bStatus=true; // boolean variable with initialization
char cOperation='+'; // character with initialization
```

Note the different styles in declaring the variables. For example, the statement

```
int nV;
```

declares an integer variable whose value is currently unknown. Such variables are called uninitialized variables. Good programming practice requires that variables be used only after they are initialized. In other words, the declaration for nV should be followed by a statement later in the program where an integer value is assigned to nV. Now consider the following declaration.

```
bool bStatus=true; // boolean variable with initialization
```

Here the boolean variable bStatus is declared and initialized with the true value. A variable must be declared once and only once before it is used in any program segment.

2.3 Expressions

An expression is made up of a sequence of tokens or basic elements that can be evaluated. For example

```
nValA + nValB
```

is an expression that is made up of two variables nValA and nValB and the addition operator, +.

Mathematical Operators

The following table lists the mathematical operators you can use in constructing an expression.

OPERATOR	MEANING
+ , -	Unary positive, negative
+	Addition
-	Subtraction

*	Multiplication
/	Division
%	Modulus (or remainder)

Operator Precedence

The operators have default precedence when used in combinations. The default precedence can be overridden by using parenthesis. The order of **ascending** precedence is the following.

PRECEDENCE

0. =	Assignment
1. +	Addition
1. -	Subtraction
2. *	Multiplication
2. /	Division
2. %	Modulus
3. ()	Parenthesis
4. +, -	Unary positive, negative

Consider the following examples where integer values and arithmetic are used.

Expression	Order of evaluation	Evaluates to
5+3-2	5+3=8-2=6	6
5+3*2-1	3*2=6 ; 5+6=11-1=10	10
(5+3)*2-1	5+3=8 ; 8*2=16-1=15	15
5*6/3	5*6=30/3=10	10
5*(6/4)	6/4=1 ; 5*1=5	5
5*6/4	5*6=30/4=7	7
5%2*3	5%2=1*3=3	3

In the case of operators with the same precedence, the expressions are evaluated left to right. Note that integer arithmetic results in the fractional value being lost. $6/4$ evaluates to 1 as does $7/4$. However, with floating point arithmetic $6.0/4.0$ evaluates to 1.5. We will discuss more about this issue in Chapter 3.

Mathematical Functions

The following table lists the commonly used mathematical functions. The header file `<math.h>` or `<cmath>` needs to be included before we can use these functions. The function parameters and computed value are of `double` data type, unless otherwise noted.

Name	Function	Examples

<code>sin(x)</code>	Computes sine of an angle x expressed in radians. Similarly <code>cos(x)</code> and <code>tan(x)</code> compute cosine and tangent values respectively.	<code>sin (dX)</code> <code>cos(dX*dY/2.0)</code> <code>tan(2.3+4.5/dZ)</code>
<code>asin(x)</code>	Arc sine. Function returns angle expressed in radians. Similarly <code>acos</code> and <code>atan</code> are functions for arc cosine and arc tangent respectively.	<code>asin(dB)</code>
<code>cosh(x)</code>	Hyperbolic cosine. Similarly <code>sinh</code> and <code>tanh</code> are hyperbolic sine and tangent respectively.	<code>cosh (dFill)</code>
<code>sqrt (x)</code>	Square root of x.	<code>sqrt(25.0+dAlpha)</code>
<code>pow (x,y)</code>	x raised to power y (x^y). This is an overloaded function.	<code>pow (fX, 3.5f)</code>
<code>log(x)</code>	Natural logarithm (base e) of x.	<code>log (fImpulse)</code>
<code>log10(x)</code>	Logarithm to base 10 of x.	<code>log10 (2.5)</code>
<code>fabs(x)</code>	Absolute value of x.	<code>fabs (dQ-dR)</code>
<code>abs(x)</code>	Absolute value of x.	<code>abs(nM)</code>
<code>exp(x)</code>	Computes e to the power of x.	<code>exp (fabs(dA)+dB)</code>
<code>ceil (x)</code>	Returns the smallest integer that is greater than or equal to x.	<code>ceil(2.9) is 3.0</code>
<code>floor(x)</code>	Returns the largest integer that is less than or equal to x.	<code>floor(2.9) is 2.0</code>

Here are more examples of expressions.

Mathematical expression	C++ expression
$\frac{bh^3}{12}$	<code>fB*pow(fH, 3.0)/12.0</code>
$\frac{ML^2}{2EI}$	<code>(fM*fL*fL)/(2.0*fE*fI)</code>

$\frac{\sin(a)\cos(b)}{\sqrt{1+c^2}} - 5.5$	<code>(sin(fA)*cos(fB))/sqrt(1.0+fC*fC)-5.5</code>
$y x - \log(a) $	<code>fY*fabs(fX-log(fA))</code>
$10t \exp(-t)$	<code>10.0*fT*exp(-fT)</code>
$\frac{1}{2} + \sin^{-1}\left(\frac{a}{\pi}\right)$	<code>0.5+asin(fA/3.1415926)</code>

2.4 Assignment Statements

There are several types of C++ statements including an assignment statement. An example of an assignment operator is the simple assignment (`=`) operator, which assigns the value of its right operand to its left operand. For example

```
nValue = nValA;
```

is an assignment statement. An assignment statement is usually composed of a destination variable, an assignment operator, and an expression to be assigned. In the following example

```
nValue = nValA + nValB;
```

the values of variables `nValA` and `nValB` are added together and the variable `nValue` is assigned the result of the addition operation. Do not be surprised if you find that a variable appear on both sides of the assignment operator. For example the statement

```
nA = nA + 1;
```

implies the following – take the current values of `nA`, add 1 to it, take the result and assign it as the new value of `nA`. The value of the destination variable is called **l-value** (appearing on the left side of the equality) and the value of the expression on the right side of the equality is called the **r-value**.

There are several arithmetic assignment operators that can be used to abbreviate assignment expressions. These operators can reduce any statement of the form

```
variable = variable operator expression;
```

where operator is one of the mathematical operators (`+`, `-`, `*`, `/`, or `%`), to the form

```
variable operator = expression;
```

The following table shows the arithmetic assignment operators, sample statements, and their equivalent statements.

Assignment operator	Sample C++ statement	Equivalent C++ statement
<code>+=</code>	<code>c += 9</code>	<code>c = c + 9</code>
<code>-=</code>	<code>d -= 5</code>	<code>d = d - 5</code>
<code>*=</code>	<code>e *= 2</code>	<code>e = e * 2</code>
<code>/=</code>	<code>f /= 3</code>	<code>f = f / 3</code>
<code>%=</code>	<code>g %= 7</code>	<code>g = g % 7</code>

If `nA`, `nB` and `nC` are all declared to be integers and `fP`, `fQ` and `fR` are declared to be floats, then the following assignment statements have the same data types on both sides of the assignment operator.

```
nA = 10 + nB*nC;
fP = 10.2f*fQ - fR;
```

What does the compiler do when it encounters an assignment with different data types? For example, how is

```
dX = 10 + 2.5*nA;
```

evaluated where the left hand side is a double variable (`dX`) and the expression on the right is made up of an integer constant (10), a double constant (2.5) and an integer variable `nA`? We will look at this situation in greater detail in the later chapters.

2.5 Simple Input and Output

The C++ standard libraries are rich in supporting input/output operations. In this section we will introduce the stream concept and discuss the most common I/O operations.

Streams

A **stream** is a sequence of bytes. In input stream, the bytes flow from a device like keyboard or disk drive to main memory. In output stream, the bytes flow from main memory to a device such as display screen, printer, or disk drive.

The **iostream** header file contains basic information required for all stream I/O operations. This header file contains objects such as `cin`, `cout`, `cerr`, and `clog`, some of which will be discussed here.

The **iostream** library contains many classes that handle I/O operations. The **istream** class supports stream input operations. The **ostream** class supports stream output operations. The **iostream** class supports both the input and output operations. The **iostream** class is derived from both the **istream** and **ostream** classes.

Stream Input

Stream input may be performed with the right shift operator (`>>`), also referred to as the **stream-extraction operator**. This operator normally skips whitespace characters like blanks, tabs, and newlines and returns zero (false) when end of file is encountered in the input stream.

cin is an object of the `istream` class and corresponds to standard input device. In the following, the `cin` object used with the stream-extraction operator causes a value for integer variable `nScore` to be input from `cin` to memory. Assume that `nScore` has previously been declared an integer. Then the statement

```
cin >> nScore;
```

is used to read in the value of `nScore`.

Stream Output

The `ostream` class offers several output capabilities. These include output of standard data types with the stream-insertion operator, characters with the `put` member function², unformatted output with the `write` member function, and various formatted output.

Stream output may be performed with the left shift operator (`<<`) which is also referred to as the **stream-insertion operator**.

cout is an object of the `ostream` class and it corresponds to the standard output device. The `cout` keyword has the following syntax.

```
cout << ...;
```

where the insertion operator `<<` is used to output standard types represented above as Consider the following example.

```
int nA=1, nB=10, nC;
nC = nA + nB;
cout << "The sum of " << nA << " and " << nB << " is " << nC
     << "." << endl;
```

Note the use of `endl` (end line) stream manipulator. The `endl` stream manipulator creates the same result as `\n` escape sequence and also causes the output buffer to be output immediately even if it not full. The output generated by the above statement is shown below.

The sum of 1 and 10 is 11.

For example, to read an integer value we could use the following statements.

```
#include <iostream>           // iostream class
using std::cin;               // standard input
using std::cout;              // standard output
...
...
int nScore;
...
cout << "What is the score? ";// ask user for the score
cin >> nScore;                // read the user input
...
```

² Member functions will be discussed with objects and classes starting in Chapter 7.

Formatting with Stream Manipulators

As mentioned before, C++ provides many capabilities to format input and output data. Stream manipulators perform the formatting tasks. The most common stream manipulators, descriptions, and example are in the following lists.

Floating-Point Precision

setprecision() controls the number of significant digits or significant decimal digits with one integer argument
precision() same as above and returns the current precision setting with no argument (precision 0 restores the default precision value 6 for both)

Field Width

width() sets the field width and returns the previous width with one integer argument, and with no argument returns the current setting
setw() sets field width (a value wider than the field width will not be truncated and width setting applies only for the next insertion or extraction)

The **precision** and **setprecision** manipulators control the output. If the display format is scientific or fixed, then the precision indicates the number of digits after the decimal point. If the format is automatic (neither floating point nor fixed), then the precision indicates the total number of significant digits. This setting remains in effect until the next change. The **width** and **setw** manipulators control how many character spaces are reserved for outputting a value.

Here are a few example statements showing how the precision and field width can be controlled. We will examine these features in greater detail in the following chapters.

```
cout.precision (10);
cout << setprecision(10);    // same effect as previous line
cout.width (20);
cout << setw(20);           // same effect as previous line
```

Formatted output is shown in the examples that follow.

Example Program 2.5.1 Data types with formatted output

In this example we will look at computations involving different data types. The program is written to compute through the use of variables of different data types, the following expressions.

integer:	100000 + 200000
long:	(-64000) × (-12800)
float:	$\frac{1}{3}$ $\frac{1}{16}$
double:	$\frac{1}{3}$ $\frac{1}{16}$

main.cpp

```

1 #include <iostream> // header file
2 using std::cout; // standard output class
3 using std::endl; // standard newline character
4
5 #include <iomanip> // header file
6 using std::setprecision; // set precision
7 using std::setw; // set width
8
9 int main()
10 {
11
12     // illustrate integer arithmetic
13     int nA, nB, nC;
14     nA = 100000; nB = 200000;
15     nC = nA + nB;
16     cout << " int: " << nA << " + " << nB << " = " << nC << endl;
17
18     long lA, lB, lC;
19     lA = -64000L; lB = -12800L;
20     lC = lA * lB;
21     cout << "long: " << lA << " * " << lB << " = " << lC << endl;
22
23     // illustrate floating point arithmetic
24     cout << setprecision(12); // 12 decimal places
25     float fX, fY, fZ;
26     fX = 1.0f/3.0f; fY = 1.0f/16.0f;
27     fZ = fX/fY;
28     cout << " float: " << setw(20) << fX << " / " << setw(10)
29         << fY << " = " << setw(20) << fZ << endl;
30
31     cout << setprecision(15); // 15 significant digits
32     double dX, dY, dZ;
33     dX = 1.0/3.0; dY = 1.0/16.0;
34     dZ = dX/dY;
35     cout << "double: " << setw(20) << dX << " / " << setw(10)
36         << dY << " = " << setw(20) << dZ << endl;
37
38     // all done
39     return (0);
40 }
```

As before, we see examples of function prototyping using the header files `iostream` and `iomanip`. The output generated by the program is shown in Fig. 2.5.1. Note the difference in the output between the float and the double outputs. The float value beyond 6 significant digits is unreliable.

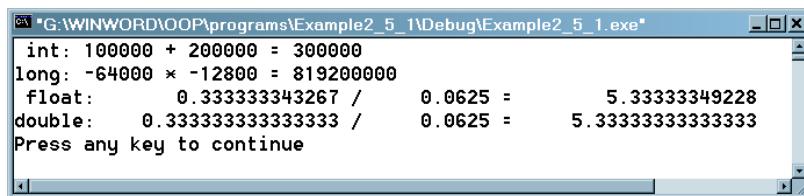


Fig. 2.5.1 Output generated by Microsoft Visual Studio for Example 2.5.1

A few points to note. Multiple statements can be typed on a single line as long as they are separated by semicolons – see lines 14, 26, and 33. A complete statement can appear on more than one line – see lines 28 and 29, and 35 and 36.

Example Program 2.5.2 Math function and simple I/O

In this example we will look at using the inbuilt math functions. We will write a program to ask the user for an angle (in degrees) and compute the sine of that specified angle.

main.cpp

```

1  #include <iostream> // header file
2  using std::cin;    // standard input class
3  using std::cout;   // standard output class
4  using std::endl;   // standard newline character
5
6  #include <cmath>    // contains math function declarations
7
8  int main()
9  {
10
11     // conversion values
12     const double PI = 3.1415926, ANGLETORADIANS = PI/180.0;
13     double dAngle;    // to store the angle
14
15     // compute sin of an angle
16     cout << "Input an angle in degrees: ";
17     cin >> dAngle;
18     cout << "Sine of " << dAngle << " degrees is: "
19           << sin(dAngle*ANGLETORADIANS) << endl;
20
21     // all done
22     return (0);
23 }
```

The `const` qualifier before a variable signifies that the value of the variable cannot change over the course of the program. If an attempt is made to change the value of the variable, a compiler error results. Consider the following statements.

```
const int NVALUES = 3; // NVALUES is declared as a const int
NVALUES = NVALUES + 3; // invalid statement. will not compile
```

This is a defensive programming mechanism and one should use such a programming style. A sample output generated by the program is shown in Fig. 2.5.2.

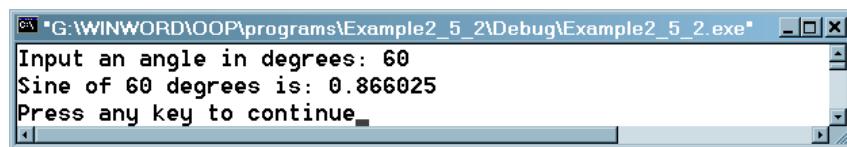


Fig. 2.5.2 Output generated by Microsoft Visual Studio for Example 2.5.2

2.6 Vector Variables

So far we have looked at scalar data types – data types that store one value. What if we wish to store three floating point values representing the (x, y, z) coordinates of a point in a single variable. C++ provides vector variables for such a purpose. The correct syntax for declaring a vector variable is as follows.

```
datatype variablename[integer constant];
```

For example, to store the coordinates of a point we have two options shown below.

```
float fXCoor, fYCoor, fZCoor; // 3 different scalar variables
float fVCoordinates[3];       // 1 vector variable
```

The three values in the vector can be accessed using the [] operator as follows. `fVCoordinates[0]` refers to the first value in the `fVCoordinates` vector, and `fVCoordinates[1]` and `fVCoordinates[2]` refer to the second and the third values, respectively. In other words, a vector of size `n` is accessed starting at index **0** through index **n-1**. This process of allocating storage space for a vector is called static allocation. We cannot dynamically allocate the storage space using the following statements.

```
int n;           // int variable. not an integer constant
cin >> n;        // obtain value of n
float fVC[n];   // illegal statement. will not compile
```

We will later see how to overcome this drawback in Chapter 8.

Example Program 2.6.1 Vector data types

We will now write a program to obtain three integer values interactively, compute their sum, and display the three numbers and their sum.

main.cpp

```
1 #include <iostream> // header file
2 using std::cin;    // standard input class
3 using std::cout;   // standard output class
4 using std::endl;   // standard newline character
5
6 int main()
7 {
8
9     const int MAXINPUT = 3;      // maximum number of input values
10    int nVNumbers[MAXINPUT];    // vector of numbers
11    int nSum;                  // to store the sum of all numbers
12
13    // get the user input
14    cout << " Input the first number: "; cin >> nVNumbers[0];
```

```

15     cout << "Input the second number: "; cin >> nVNumbers[1];
16     cout << " Input the third number: "; cin >> nVNumbers[2];
17
18     // compute the sum
19     nSum = nVNumbers[0] + nVNumbers[1] + nVNumbers[2];
20
21     // display the output
22     cout << "The sum of " << nVNumbers[0] << endl
23         << "           " << nVNumbers[1] << endl
24         << "           " << nVNumbers[2] << endl
25         << "           is: " << nSum << endl;
26
27     // all done
28     return (0);
29 }
```

As a matter of programming style, it is desirable to declare constant integer variables than use an integer constant throughout the program. First, it is easier to understand the significance of MAXINPUT than the number 3. Second, if the value of the constant needs to be modified, then only one statement needs to be changed in the program as opposed to changing the integer constant wherever it is used in the program.

A sample output generated by the execution of the program is shown in Fig. 2.6.1.

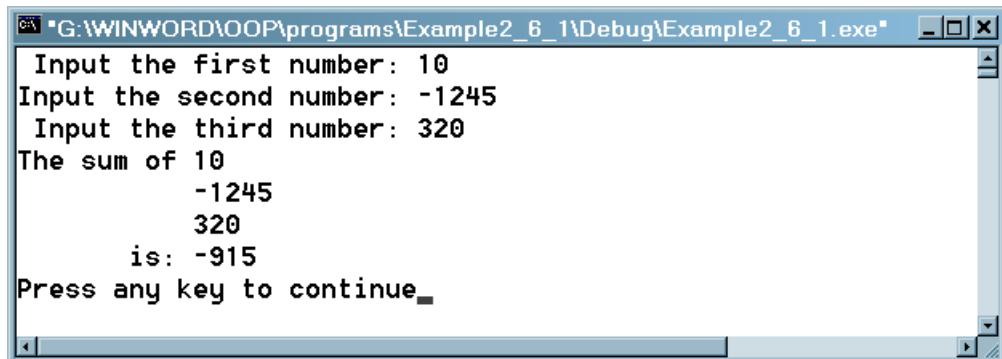


Fig. 2.6.1 Output generated by Microsoft Visual Studio for Example 2.6.1

Tip: Here is a common programming error. What will happen if we use nVNumbers [3] to access the last value in the above computer program? The situation is unpredictable. We will see how to trap and correct such errors in Chapter 9.

We can also store a string of characters in a `char` vector.

```
char szHeader[] = "Welcome to my 4-function calculator program";
```

Note that the above statement defines the variable `szHeader` as a character string and initializes the value of the variable to the specified string. The compiler automatically computes the length of the vector needed to store the string. Character strings **require an additional storage space** to store the

string delimiter – a special character that signals the end of the character vector. This character is ‘\0’. Hence the other term for such strings – a zero-terminated string. Consider the following example.

```
char szFirstName[4] = "John"; // incorrect
```

This is invalid since the string John needs five spaces not four. The fifth space is required to store the string delimiter. String manipulation can become very cumbersome. In the following statements, while the variable declaration is legal, the assignment is not.

```
char szMovieTitle[18];
szMovieTitle = "Lord of the Rings"; // will not compile
```

However, the following statements are valid.

```
szMovieTitle[0] = 'L';
szMovieTitle[1] = 'o';
```

We will use the standard `string` class whenever possible. This class is gradually introduced in the following chapters and is discussed in detail in Chapter 7.

2.7 Troubleshooting

To err is human, and most IDE’s provide tools to provide assistance in finding and correcting the errors. In this section we will look at how some of these tools can be used.

Example Program 2.7.1 Compile errors (Example Program 2.1.1 revisited)

It is quite frustrating as a beginning programmer to find that seemingly small issues can prevent a program from compiling, or linking or executing. In this section, we will see how to react to error messages.

main.cpp

```

1  /* EXAMPLE 2.7.1
2   Copyright(c) 2001-09, S. D. Rajan
3   Object-Oriented Numerical Analysis
4
5   OBJECTIVES
6   (1) Learning to troubleshoot
7   (2) Compile error messages
8
9   */
10
11 #include <iostream> // header file
12 using std::cout // standard output class
13
14 int main()
15 {
16     cout << "Welcome to Object-Oriented Numerical Analysis.\n";
17 }
```

```
18     rwtturn 0;
19 }
```

Compiling the program yields the following error messages.

```
-----Configuration: Example2_7_1 - Win32 Debug-----
Compiling...
main.cpp
g:\winword\oop\programs\example2_7_1\main.cpp(14) : error C2144: syntax error : missing ';' before type 'int'
g:\winword\oop\programs\example2_7_1\main.cpp(14) : fatal error C1004: unexpected end of file
found
Error executing cl.exe.

main.obj - 2 error(s), 0 warning(s)
```

The first error message is relatively simple. A syntax error is detected in line 14 – a semicolon is missing and the compiler expects to see a semicolon before the keyword `int`. The program is missing a semicolon in line 12. The line should have been typed as

```
using std::cout;
```

The second error message is more cryptic. The compiler encounters the first error, does not know how to correct the error but still moves on to compile the rest of the program. The ‘unexpected end of file found’ error is because of the first error. The program has one more error that goes undetected. If we correct line 12 and recompile the program, we get the following error messages.

```
-----Configuration: Example2_7_1 - Win32 Debug-----
Compiling...
main.cpp
G:\WINWORD\OOP\programs\Example2_7_1\main.cpp(18) : error C2065: 'rwtturn' : undeclared
identifier
G:\WINWORD\OOP\programs\Example2_7_1\main.cpp(18) : error C2143: syntax error : missing ';' before 'constant'
G:\WINWORD\OOP\programs\Example2_7_1\main.cpp(19) : warning C4508: 'main' : function should
return a value; 'void' return type assumed
Error executing cl.exe.

main.obj - 2 error(s), 1 warning(s)
```

Once again, because of a simple typing error, we see several error messages. As the first error message states, the error is in line 18. The correct statement should have been

```
return 0;
```

The second and the third messages are displayed because the compiler does not understand what the word `rwtturn` is.

Example Program 2.7.2 Link errors (Example Program 2.1.1 revisited)

In this example we will see what is meant by link errors.

main.cpp

```
1 /* EXAMPLE 2.7.2
```

```

2      Copyright(c) 2001-09, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5      OBJECTIVES
6      (1) Learning to troubleshoot
7      (2) Link error messages
8
9  */
10
11 #include <iostream> // header file
12 using std::cout;    // standard output class
13
14 int nain()
15 {
16     cout << "Welcome to Object-Oriented Numerical Analysis.\n";
17
18     return 0;
19 }
```

Compiling the program yields no errors. However, when we try to link the program we get the following error messages.

```

-----Configuration: Example2_7_2 - Win32 Debug-----
Compiling...
main.cpp
Linking...
LIBCD.lib(crt0.obj) : error LNK2001: unresolved external symbol _main
Debug/Example2_7_2.exe : fatal error LNK1120: 1 unresolved externals
Error executing link.exe.

Example2_7_2.exe - 2 error(s), 0 warning(s)
```

The simplified error message is “unresolved external symbol `_main`”. In plain English, the linker was looking for the main program and could not find it. Once again, we have a typo (typographic error) in the program. Line 14 should read

```
int main()
```

Syntactically there was no error in the program. We could have a function called `nain` in our program! However, as we saw early on in the chapter, every program must have one and only one `main` function. This is the function where the execution of the program begins. There is ambiguity as to where the program should start its execution if this function does not exist.

Example Program 2.7.3 Logical error (Example Program 2.5.2 revisited)

Finally, we will see an example of a logical error. These errors are the most difficult to find and correct.

main.cpp

```

1  /* EXAMPLE 2.7.3
2  Copyright(c) 2001-09, S. D. Rajan
3  Object-Oriented Numerical Analysis
4
```

```

5      OBJECTIVES
6          (1) Consequences of bad programming habits.
7          (2) Debugging
8
9      */
10
11 #include <iostream>    // header file
12 #include <cmath>        // contains math functions
13
14 int main()
15 {
16
17     int Angle;    // to store the angle
18
19     // compute sin of an angle
20     std::cout << "Input an angle in degrees: ";
21     std::cin >> Angle;
22     std::cout << "Sine of " << Angle << " degrees is: "
23             << sin(Angle) << '\n';
24
25     // all done
26     return (0);
27 }

```

The above program compiles and links correctly. However it does not produce the correct answer.

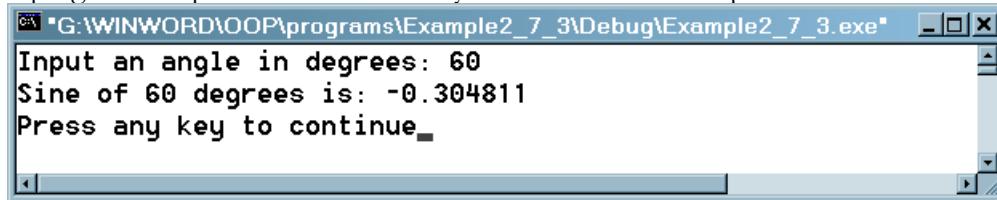


Fig. 2.7.1 Invalid output generated by the program

The program has a logical error. Logical errors are more difficult to detect and correct. The program has two errors. First, the variable to store the angle, `Angle`, is declared an integer. It should be declared a floating point variable. Second, the conversion from degrees to radians is not made before the `sin` function is used.

Most computer systems provide an interactive debugger to help the programmer control the execution of the program and hence locate the source of logical errors in the program. Interactive debuggers are useful only if you have a debugging strategy. You should know what to expect from the program (the correct output) and use the debugger to find out why the program does not compute the correct results. Creating a “hand solution” is the first step before you using the debugger. For example, we know that $\sin(25.6^\circ) = 0.432086$.

Some of the features provided by interactive debuggers include the following.

- Setting breakpoints: A breakpoint is a program location (or statement). The debugger suspends program execution when (and if) that location is reached. At this stage the user can carry out a variety of tasks such as examining or even changing the current values of certain key variables. If necessary, a user can set or remove one or more breakpoints spread throughout the program.

- Execution flow control: The user can step through the program one statement at a time, step over functions, or can continue execution till the next breakpoint is reached.
- Examine and change values of variables: The user can examine the current values of variables and if necessary, change the values of these variables.
- Set watchpoints: The user can use this feature to find where in the program the value of a certain variable is changed etc.

Let us set up the strategy to debug Example 2.7.3. First we will set up a breakpoint at line 22-23. The reason is that we want to ensure that the program reads our input correctly and stores the correct value in the variable `Angle`. Second, when the program execution encounters the breakpoint, we will examine the value of `Angle`.

Below we present the output generated by Microsoft Visual Studio when the breakpoint is encountered. The program output window is shown in Fig. 2.7.2. The IDE main screen is shown in Fig. 2.7.3.

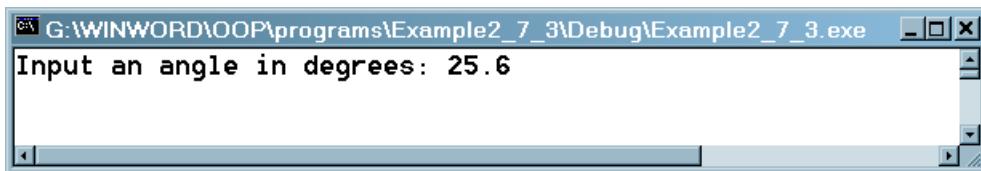


Fig. 2.7.2 Program output window. Program is in suspended state of execution.

Note that the value of `Angle` as displayed in one of the windows is 25 not 25.6 as it should be. This should alert us to the fact that the fractional component of the input value is being truncated. If we look at the source code, we will realize that `Angle` is declared as an integer not a floating point variable.

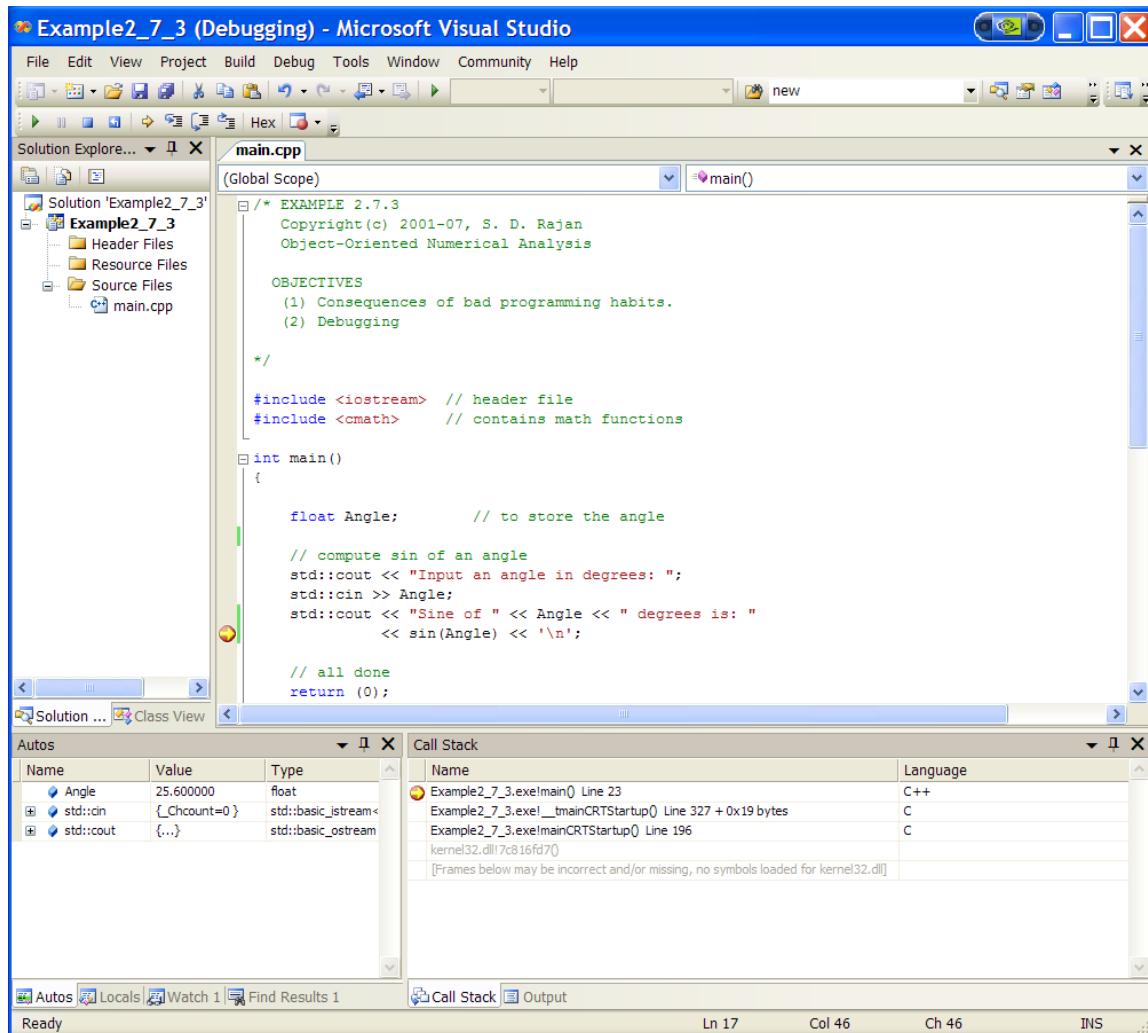


Fig. 2.7.3 Debugging screen. Program is in suspended state of execution.

We will make a few changes to the program and start the debugging process once again. The changes are as follows.

```
float Angle;          // to store the angle
float SineAngle;      // to store sine of the angle

// compute sin of an angle
cout << "Input an angle in degrees: ";
cin >> Angle;
SineAngle = sin(Angle);
cout << "Sine of " << Angle << " degrees is: "
     << SineAngle << endl;
```

The second part of the error is a little more difficult to debug.

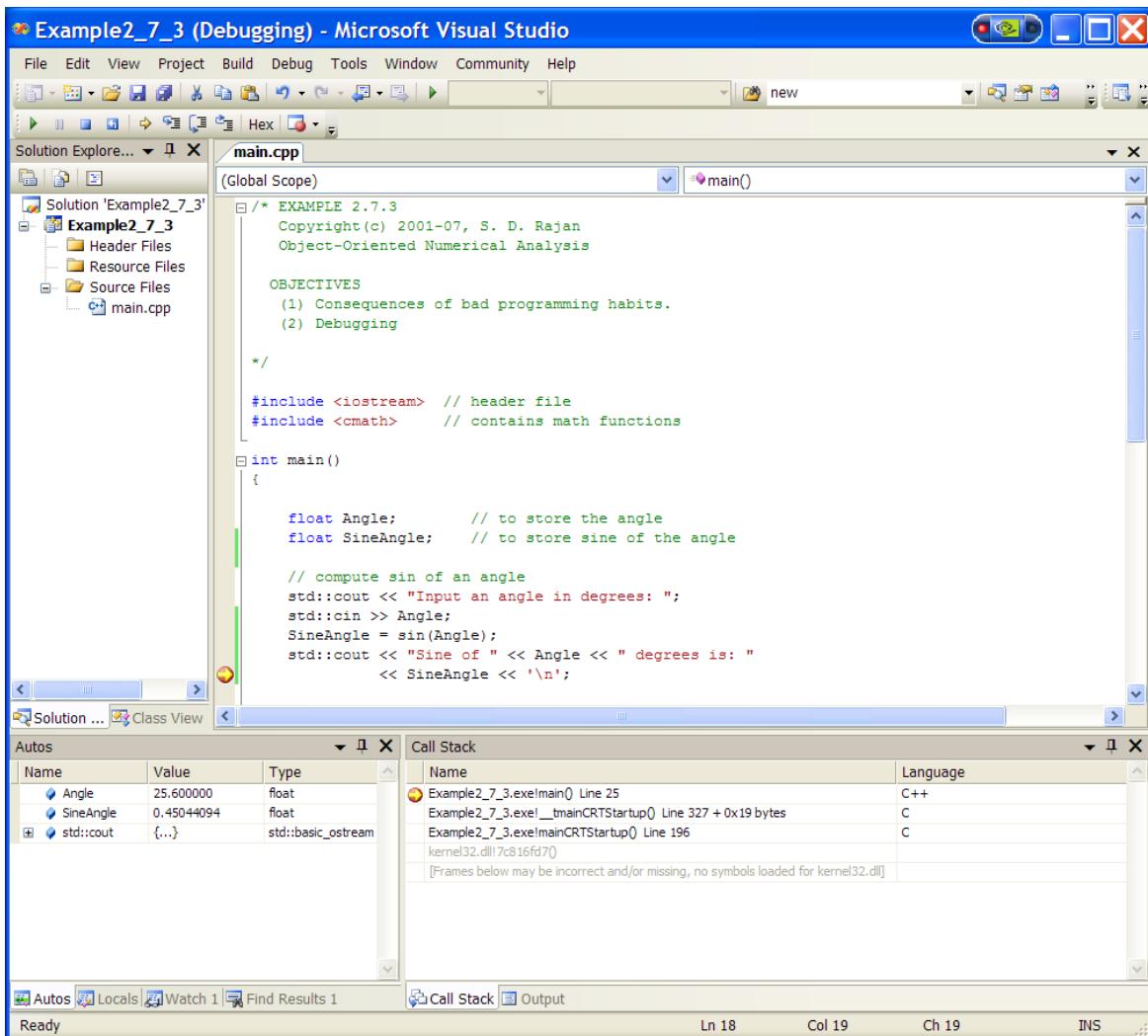
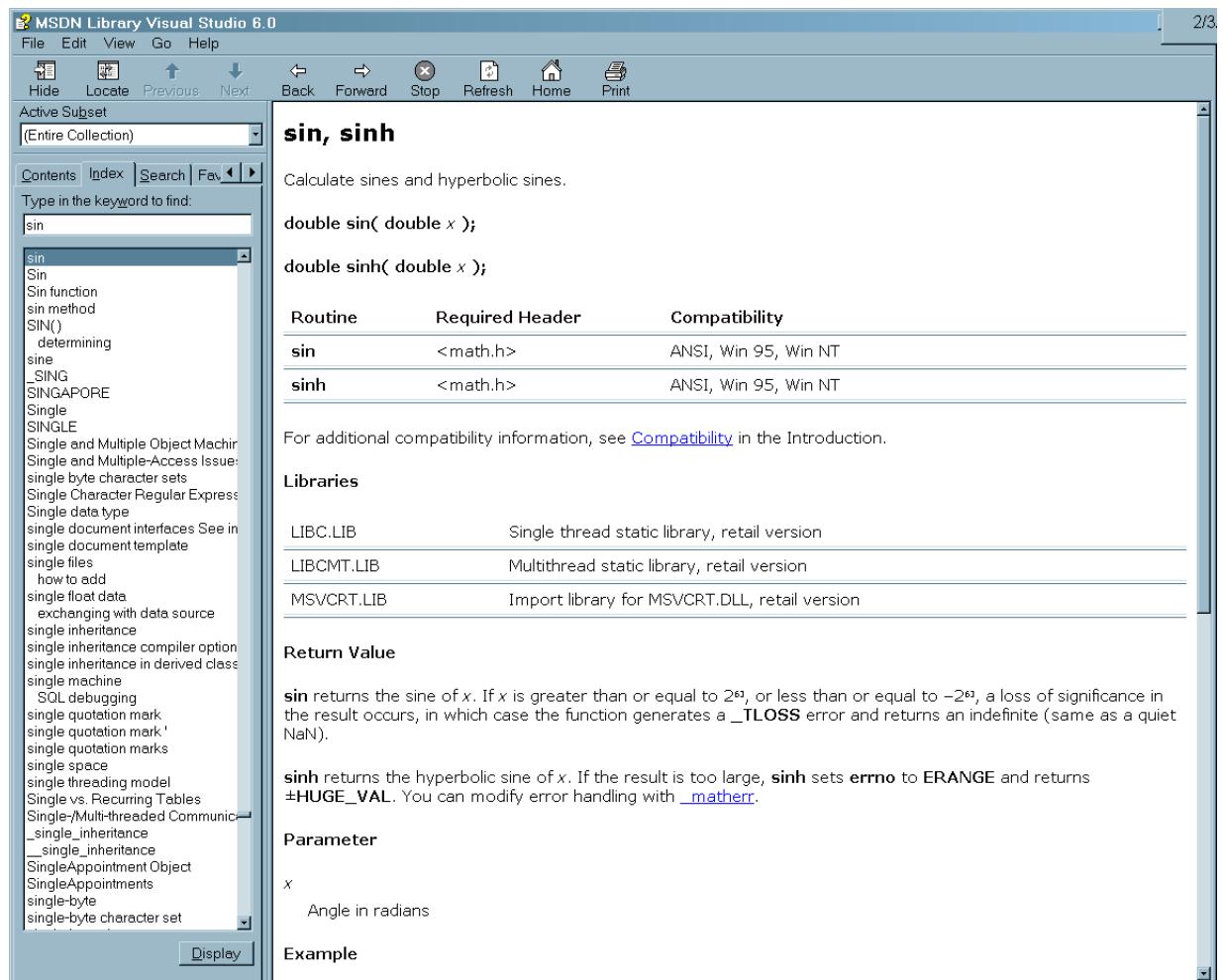


Fig. 2.7.4 Debugging screen. Program is in suspended state of execution.

We set a second breakpoint and the output (at the second breakpoint) is shown in Fig. 2.7.4. The evaluated value of $\sin(25.6^\circ)$ is shown as 0.45044094. What is wrong? At this stage, we should look at the C++ documentation on the `sin(x)` function. The help facility for the `sin` function is shown below and we immediately discover that the parameter `x` should be in radians (Angle in radians).

Fig. 2.7.5 Microsoft Visual Studio on-line help documentation on the use of **sin(x)** function

Remember that debugging is an art as much as it is a science.

2.8 The namespace Concept

Some readers must have noticed that while we have used the combination

```
using std::cout; // defined at the beginning of the file  
....  
    cout << "What is n? ";
```

we have also used (see Example 2.7.3 where `using std::cout` is not defined)

```
std::cout << "What is n? ";
```

Both these usages are correct. In fact there is a third usage style.

```
using namespace std; // defined at the beginning of the file  
....  
    cout << "What is n? ";  
    cin >> n;
```

The `using` syntax does not require that keywords such as `cout`, `cin` etc. be qualified with the `std::` qualifier. The prefix `std::` indicates that the keywords `cout`, `endl`, `setw` etc. are defined inside the namespace `std`. The standard namespace `std` is defined in the standard library and is available by inclusion of the appropriate header file (e.g. `#include <iostream>`). Other namespaces can be defined and used by the programmer and the `::` operator (scope operator). This means that two identical keywords with possibly different functionalities can be used in the same program segment as long as their usage is properly defined in both namespaces and they are properly referenced in the program that uses them.

As a matter of style, we will usually not employ the `using` keyword in the rest of the text and instead use the `std::` qualifier when appropriate.

Interested readers are urged to look at Example 2_8_1 for a user-defined namespace example.

Summary

In this chapter we saw how to write, compile, link, execute and debug simple C++ programs. In the following chapters will we learn more about C++. We will then look at how to get organized and develop the program.

Below we summarize the important facts learnt in this chapter.

- Definitions and declarations must precede usage. Look at C++ compiler having access to a program dictionary that has three components – (a) C++ keywords and tokens, (b) functions whose prototypes are available in header files, and (c) user-defined variables and functions. Imagine that this is a dynamic dictionary with respect to (c). In other words, items may be added to the dictionary while the compiler is interpreting the program. If something is used in the program that cannot be found in the dictionary, then the compiler issues an error message.
- Every program has one and only one `main` function. This is the location from where the execution of the program starts.
- Comments in programs start with the pair `/*` and end with the pair `*/`. Comments on a single line occur to the right of the pair `//`.
- A block of statements occur within the braces `{` and `}`. As an example, the statements in the `main` function can be found within the braces. A semicolon `;` is used to terminate most commonly used C++ statements.
- Usually there are two types of files found in C++ programs – `.cpp` and `.h` files. Usually the `.cpp` files contain the statements that are executed, and the `.h` files contain the definitions and declarations. Note that the file extensions may be different with different compilers and IDEs.
- The more commonly used standard data types are `short`, `int`, `long`, `float`, `double`, `bool` and `char`.
- Variables are used to store values of the standard data types. Variables are identified by variable names.
- Scalar variables store one value. Vector variables store several values.
- Functions are independent program segments that can be called from different locations in a program. Functions are discussed later starting in Chapter 4.
- C++ provides several functions including mathematical that the programmer can use in his or her program.

- Expressions can be created using constants, variables, functions and operators. Expressions with mathematical operators follow certain evaluation rules.
- There are several types of C++ statements such as assignment, input/output, etc.
- C++ provides streams (sequence of bytes) for obtaining input from devices such as a keyboard or disk, and for outputting data to a display device, printer, or disk.

EXERCISES

One of the common errors in writing the computer programs suitable for the topics discussed in this chapter is deciding what data type should be used for the variables. Spend some time to think over the problem before deciding the data type. For example, should the sides of a rectangle be represented by an integer or a floating point variable?

Appetizers

Problem 2.1

Write a program to output the following pattern on the screen.

```
***  
****  
***
```

Problem 2.2

Write a program to interactively obtain an integer value and display (a) the negative of that number, (b) its absolute value, and (c) its square.

Problem 2.3

Write a program to interactively obtain a floating point value, x for each of the three following cases and display (a) the square root of that number, (b) its absolute value, and (c) 4.5^x .

Problem 2.4

For the following values: $a = 1.2$, $b = -35.6$, $c = 1056.78$, $d = -22.5^\circ$, $e = 153.4^\circ$, write a program to compute and display the results of the following expressions: (a) $\frac{a+b}{c}$, (b) $\sqrt{\frac{|a+b|}{c-33.3}}$, (c) $a + \frac{\sqrt{c+b^2}}{a^{0.4}}$, (d) $\sin(a) + \cos\left(\frac{c}{(a+b)^2}\right)$, (e) $\frac{\tan(d)}{\tan(d)-\sin(e)}$.

Main Course

Problem 2.5

Write a program to obtain the length and width of a rectangle, and compute and display the perimeter and the area of the rectangle.

Problem 2.6

Write a program to carry out the following conversions (a) obtain the length in *in* and convert it to *m*, (b) obtain the temperature in $^{\circ}F$ and convert it to $^{\circ}C$, and (c) obtain the mass in *slg* and convert it to *kg*.

Problem 2.7

Write a program to obtain the (x, y, z) coordinates of two points. Now compute and display (a) the distance between the two points, and (b) a unit vector from point 1 to point 2 (display the unit vector as $a\hat{i} + b\hat{j} + c\hat{k}$).

Problem 2.8

Write a program to obtain the following material values expressed in $(in, ^{\circ}F, slg, lb)$ and convert them to $(m, ^{\circ}C, kg, N)$ - (a) stress, (b) coefficient of thermal expansion, and (c) mass density.

C++ Concepts*Problem 2.9*

Write a program to display the values of the function $y(x) = ax^3 + bx^2 + cx + d$ for the range $-10 \leq x \leq 15$ using an increment of 5. Obtain the values of the coefficients of the cubic polynomial from the user. Display the values as follows (one per line).

```
(x, y(x)) = (x value, y value)
```

For example, $(x, y(x)) = (-5.0, -120.3)$

Problem 2.10

Summarize all the facts about C++ that (a) you have learnt from this chapter, and (b) from reference material outside this book.

Chapter 3

Control Structures

“When you get to the fork in the road, take it.” Yogi Berra

“If you do not know where you are going, you’ll wind up somewhere else.” Yogi Berra

“Here are three great questions which in life we have over and over again to answer: Is it right or wrong? Is it true or false? Is it beautiful or ugly? Our education ought to help us to answer these questions.” John Lubbock

“When I turned two I was really anxious, because I’d doubled my age in a year. I thought, if this keeps up, by the time I’m six I’ll be ninety.” Steven Wright

“To climb steep hills requires slow pace at first.” William Shakespeare

Now that we know how to write simple C++ programs, it is but natural to ask “How can I write useful programs?” As we saw in Chapter 1, computer programs manipulate information. The algorithms that are used in manipulating the information (a) test conditions that are to be met for certain actions to take place, and (b) repeatedly execute a number of steps for a specified number of times or until certain conditions are met.

Consider a simple example of a vector that contains integer values, e.g. -20, 15, 20, 55, -130. We need to develop an algorithm that will search this vector looking for a known target value and report whether the integer values contain the target or not, and if the target is located, where in the vector the target is located. We could use the following algorithm to solve this problem.

Input: Target value, t , and the vector of integer numbers, V .

Output: The location, l where the target t exists in V .

Step 1: Find how many elements, n in the vector V .

Step 2: Loop through all the elements starting at the first location, i.e. $i = 1, 2, \dots, n$.

Step 3: Compare the number $V(i)$ against the target t . If they are equal, set $l = i$. Exit.

Step 4: Increment $i = i + 1$.

Step 5: If $i \leq n$, go to Step 2. Else set $l = 0$ indicating that the target was not found. Exit.

An examination of the algorithm shows that repetition takes place in Steps 2 through 5 and that conditional tests occur in Steps 3 and 5. The conditional tests are necessary not only to find the target but also to terminate the repeated execution of the Steps 2 through 5. In the rest of the chapter we will see how to use C++ control structures.

Objectives

- To understand the concept of control structures.
- To understand and practice **selection** concept.
- To understand and practice **repetition** concept.

3.1 Selection

Quite often, decisions have to be made in any algorithm. Some steps in an algorithm may be needed only if certain conditions are met. The selection concept can be used to handle such a situation. C++ provides two constructs where selection is possible – through the use of **if .. else** and the use of **switch** statements. We will look at the **if .. else** usage first.

if .. else syntax

The general syntax can take on several forms as shown below.

Form 1

```
if (expression)
    statement(s) if expression is true;
```

Form 2

```
if (expression)
    statement(s) if expression is true;
else
    statements(s) if expression is false;
```

Form 3

```
if (expression1)
    statement(s) if expression1 is true;
else if (expression2)
    statements(s) if expression2 is true;
else if (expression3)
    statements(s) if expression3 is true;
...
else
    statements(s);
```

Form 4

```
if (expression1)
    statement(s) if expression1 is true;
else if (expression2)
    statements(s) if expression2 is true;
else if (expression3)
    statements(s) if expression3 is true;
```

Each expression captures a condition that needs to be met and the statements that follow specify the action that needs to be carried out if the condition is true. At most, only one of the expressions is evaluated as true and the statements associated with the action are executed. As we can see by the different forms, the **else** part of the statement is optional.

When more than one statement is associated with any part of the construct, these statements must be enclosed within the {} braces. The expression used in a selection statement must evaluate as **true** or

`false` and can be made up of logical or relational operators. C++ considers 0 to be `false` and a non-zero value to be `true`.

Relational Operators: The most commonly used relational operators are `<` (less than), `<=` (less than or equal to), `>` (greater than), `>=` (greater than or equal to), `!=` (not equal to), and `==` (equal to). Let us look at some examples.

The statements “If the score in the exam is greater than or equal to 60 then the student has passed the exam. Otherwise the student has failed the exam.” can be implemented as

```
if (nScore >= 60)
    cout << "Passed the exam.";
else
    cout << "Failed the exam.";
```

The statement “If the number of entries is not equal to zero, then the average of all the entries is the sum over the number of entries” can be implemented as

```
if (nEntries != 0)
{
    fAvg = fSum/nEntries;
    cout << "Average of the " << nEntries <<
        " numbers is " << fAvg;
}
else
    cout << "Average cannot be computed.;"
```

The relational operators are used to compare two quantities that must be of the same data type, or must be such that a suitable conversion is available to facilitate the comparison of the two quantities. The previous example can also be written as

```
if (nEntries == 0)
    cout << "Average cannot be computed.";
else
{
    fAvg = fSum/nEntries;
    cout << "Average of the " << nEntries <<
        " numbers is " << fAvg;
}
```

Tip: It is a common programming error to use the assignment operator `=` instead of using equality operator `==` in a selection statement.

For example

```
if (nA == 5)
```

is not the same as

```
if (nA = 5)
```

However, both the statements will compile. The danger is that the second form will always evaluate as true, an unintended consequence.

Logical Operators: More complex selections can be made by using logical operators. The most commonly used operators are || (OR) and && (AND). In a typical usage we have a compound expression in the following forms

```
if (expression1 || expression2) ...
if (expression1 && expression2) ...
```

where the selection is made up of two expressions and involve either OR or AND operators. The final evaluation of such an expression is shown below.

Expression 1	Operator	Expression 2	Evaluates to
true	and	true	true
false	and	true	false
true	and	false	false
false	and	false	false
true	or	true	true
true	or	false	true
false	or	true	true
false	or	false	false

The statement “The product of two numbers is positive if both numbers are positive or both the numbers are negative” can be implemented as

```
if ((nA > 0 && nB > 0) || (nA < 0 && nB < 0))
    cout << "Product is positive.";
else
    cout << "Product is negative.";
```

The statement “A legal value of student GPA is between 0.0 and 4.0 both inclusive” can be implemented as

```
if (fGPA >= 0.0 && fGPA <= 4.0)
    cout << "Valid GPA value.";
else
    cout << "Invalid GPA value.;"
```

The statement “If the score in the math portion of the exam is greater than 90 or if the score in the language portion of the exam is greater than 90, then the student is a gifted student” can be implemented as

```
if (nScoreMath > 90 || nScoreLanguage > 90)
    cout << "Student is a gifted student.";
```

Example Program 3.1.1 Selection

We will now look at a more detailed example involving selection.

Problem Statement: Obtain the age of a person. Based on the age, classify the status of the person as follows and print the appropriate message.

Age	What to print
Between 0 and 10	Person is a child.
Between 11 and 18	Person is a juvenile.
Between 19 and 59	Person is an adult.
Greater than or equal to 60	Person is a senior citizen.

main.cpp

```
1 #include <iostream>
2 using std::cout;
3 using std::cin;
4
5 int main ()
6 {
7     int nAge;           // to store the age
8
9     cout << "Input the age: ";
10    cin >> nAge;        // get the age
11
12    if (nAge < 0) {
13        cout << "Invalid age " << nAge << ".\n";
14    }
15    else if (nAge <= 10)
16    {
17        cout << "Person " << nAge << " years old is a child.\n";
18    }
19    else if (nAge <= 18)
```

```

20      {
21          cout << "Person " << nAge << " years old is a juvenile.\n";
22      }
23  else if (nAge <= 59)
24  {
25      cout << "Person " << nAge << " years old is an adult.\n";
26  }
27  else {
28      cout << "Person " << nAge << " years old is a senior citizen.\n";
29  }
30
31 // all done
32 return (0);
33
34 }
```

In the example, all possible paths arising from the **if ... else** construct, are enclosed within the braces {}. We encourage this usage since it makes the program easier to read and makes later addition of statements less error-prone. After the age variable nAge is initialized in statement 10, the program executes sequentially. Assume that the value of nAge is 53. Execution starts at line 12, and the expression is evaluated as **false**. The next statement that is executed is 15 and the expression is evaluated as **false**. The same situation arises with statement 19. Finally, statement 23 evaluates as **true** and the associated action in line 25 is executed. The last statement that is executed in the program is line 32.

As we will see throughout the text, there is no unique way of structuring a program. We can rewrite the previous example in the following two forms.

Alternate Form 1

```

if (nAge >= 60)
    cout << "Person " << nAge << " years old is a senior citizen.\n";
else if (nAge >= 19 && nAge <= 59)
    cout << "Person " << nAge << " years old is an adult.\n";
else if (nAge >= 11 && nAge <= 18)
    cout << "Person " << nAge << " years old is a juvenile.\n";
else if (nAge >= 0 && nAge <= 10)
    cout << "Person " << nAge << " years old is a child.\n";
else if (nAge < 0)
    cout << "Invalid age " << nAge << ".\n";
```

Alternate Form 2

```

if (nAge < 0)
    cout << "Invalid age " << nAge << ".\n";
if (nAge >= 0 && nAge <= 10)
    cout << "Person " << nAge << " years old is a child.\n";
if (nAge >= 11 && nAge <= 18)
    cout << "Person " << nAge << " years old is a juvenile.\n";
if (nAge >= 19 && nAge <= 59)
    cout << "Person " << nAge << " years old is an adult.\n";
if (nAge >= 60)
    cout << "Person " << nAge << " years old is a senior citizen.\n";
```

Alternate Form 1 is easy to read (however it does not use the {} rule!). Alternate Form 2 may be easy to read but is inefficient since the conditions with all the **if** statements are tested, and is also error-prone. While a usage such as

```
if (nA) ...
```

is valid, we will avoid such usage and explicitly state our objective as

```
if (nA != 0) ...
```

Nested **if** statements

Finally a word about nested **if** statements. Consider the following statements that do not produce the desired result.

```
float fX = 4.15f, fY = 2.14f;
...
// not written correctly
if (fX <= fY)
    if (fX < sqrt(20.0))
        cout << "X is less than square root of 20.";
else
    cout << "X is greater than Y.";
```

Note that every **else** is associated with the nearest **if**. The mere fact that statements are indented does not guarantee that this association is made. The correct way to write the statements is as follows.

```
float fX = 4.15f, fY = 2.14f;
...
// corrected version
if (fX <= fY)
{
    if (fX < sqrt(20.0))
    {
        cout << "X is less than square root of 20.";
    }
}
else
{
    cout << "X is greater than Y.";
}
```

We strongly recommend that you use the {} braces to identify the block of statement(s) associated with the **if** and the **else** parts of the statement.

Conditional operator ?:

C++ provides a succinct way to take care of a specialized form of selection. For example, consider the following statements.

```
if (fX > fY)
    fA = 2.1*fB;
else
    fA = 0.5*fA;
```

The C++ conditional operator ?: can be used instead of the above statements as follows.

```
fA = (fX > fY? 2.1*fB : 0.5*fA);
```

Note that three operands are involved. The first operand captures the selection condition. If this condition is true, then the second operand is executed. If the condition is false, the third operand is executed. Here are a couple more examples.

```
fX > fY? fA=2.1*fB : fA=0.5*fA;
std::cout << (fX > fY? fA : fB);
```

3.2 Repetition

As we saw in the introductory section of this chapter, sometimes one or more statements need to be executed repeatedly until a certain condition is satisfied. The block of statements is in a loop. C++ provides three commonly used loop constructs – **while**, **do ..while** and **for** statements.

while statement

The general syntax of the **while** statement is as follows.

```
while (test condition)
{
    statement(s)
}
```

The statements within the {} are executed as long as the *test condition* expression evaluates to **true**. The test condition (or expression) is tested at the beginning of the block. The statements in the block are executed only if the expression evaluates to **true**. Hence it is possible that the statements in the block do not execute even once because the expression is not **true**.

Example Program 3.2.1 Repetition using the **while** statement

We will illustrate the usage of the **while** statement with an example.

Problem Statement: Write a program to compute the sum of the first N integers, i.e. $\sum_{i=1}^N i$.

main.cpp

```
1 #include <iostream>
2 using std::cout;
3 using std::cin;
4 using std::endl;
5
6 int main ()
7 {
8     int i=1;      // loop counter
9     int nN;       // to store N
10    int nSum=0;  // to store the sum
```

```

11
12     cout << "What is N? ";
13     cin >> nN;
14
15     while (i <= nN)
16     {
17         nSum = nSum + i;
18         i = i+1;
19     }
20
21     cout << "The sum of first " << nN << " integers is "
22         << nSum << "." << endl;
23
24     return (0);
25 }
```

Let's look at the strategy used to drive the `while` loop. A loop counter or index, `i`, is used to keep track of how many times the statements in the block need to be executed. This loop counter is initialized to 1 in line 8. If the loop counter is not initialized, the test condition cannot be evaluated correctly. The variable to store the sum, `nSum`, is initialized to zero in line 10. The test condition compares the value of `i` to `nN`. The two statements in the block execute as long as `i` is less than or equal to `nN`. Line 18 is used to increment the value of the loop counter. If this statement is omitted, the statements in the block will continue to execute forever – an infinite loop.

The test condition can be complex – we saw how complex expressions can be constructed using relational and logical operators in the previous section.

`do .. while` statement

The general syntax of the `do..while` statement is as follows.

```

do
{
    statement(s)
} while (test condition);
```

The statements within the `{}` are executed as long as the *test condition* expression remains `true`. However, unlike the `while` statement, the test condition is evaluated at the end of the block of statements not at the beginning of the block of statements. Hence the statements in the block will execute at least once. We will illustrate the usage of the `do ..while` statement with an example.

Example Program 3.2.2 Repetition using the `do ..while` statement

Problem Statement: Write a program to compute the sum of the first N integers, i.e. $\sum_{i=1}^N i$.

main.cpp

```

1 #include <iostream>
2 using std::cout;
3 using std::cin;
```

```

4   using std::endl;
5
6   int main ()
7   {
8       int i=1;      // loop counter
9       int nN;       // to store N
10      int nSum=0; // to store the sum
11
12      cout << "What is N? ";
13      cin >> nN;
14
15      do
16      {
17          nSum = nSum + i;
18          i = i+1;
19      } while (i <= nN);
20
21      cout << "The sum of first " << nN << " integers is "
22          << nSum << "." << endl;
23
24      return (0);
25  }

```

The loop counter is declared and initialized in line 8. In line 19, the loop counter is used in the test condition. Similarly, the sum is defined and initialized in line 10 and is updated in line 17.

Increment and Decrement Operators

C++ provides increment and decrement operators as follows.

++	Increment
--	Decrement

These unary operators are used in conjunction with a variable as the operand. When these variables are integers or floating point numbers, a 1 is added to or subtracted from the operand. Consider the following examples where the operators are used.

```

int nP = 12;           // nP is initialized to 12
nP++;                 // nP is now 13. same as nP = nP + 1;

int nP = 44;           // nP is initialized to 44
nP--;                 // nP is now 43. same as nP = nP - 1;

int nP = 44;           // nP is initialized to 44
--nP;                  // nP is now 43. same as nP = nP - 1;

```

When the operators are used before the operand they are known as prefix operators as in `--nP` and as postfix operators if they are used after the operand as in `nP++`. One must be careful in using these operators. Consider the case where we have a vector `nVA` of length 3 containing three values as 11, 65 and 70. The correct statements are as follows.

```

int nVA[3];
int nP = 0;           // nP is 0

```

```
nVA[nP++] = 11; // nP is 1. nVA[0] = 11
nVA[nP++] = 65; // nP is 2. nVA[1] = 65
nVA[nP] = 70; // nP is still 2. nVA[2] = 70
```

The following statements will not work.

```
int nVA[3];
int nP = 0; // nP is 0
nVA[++nP] = 11; // nP is 1. nVA[1] = 11
nVA[++nP] = 65; // nP is 2. nVA[2] = 65
nVA[nP] = 70; // nP is still 2. nVA[2] = 70
```

The difference is that when `nP++` is executed, `nP` is incremented **after** the value of `nP` is used in the expression whereas `++nP` means increment `nP` first and then use `nP` in the expression.

In the above examples, the value of the variable associated with the increment and decrement is changed by one. Finally, it should be noted that the unary increment and decrement operators can be used in more sophisticated contexts as we will see later in the text.

for statement

The general syntax of the `for` loop statement is as follows.

```
for (initialization; test condition; update)
{
    statement(s)
}
```

The basic idea is to repeatedly execute the `statement(s)` in sequence as long as the `test condition` is true. The `initialization` part is used to initialize the values of the loop control variable and if necessary, other variables. The `test condition` must evaluate to true for the statements to execute. The `update` part is executed at the end of the loop and is typically used to change the value of the loop control variable.

Example Program 3.2.3 Repetition using the `for` statement

Problem Statement: Write a program to compute the sum of the first N integers, i.e. $\sum_{i=1}^N i$.

main.cpp

```
1 #include <iostream>
2 using std::cout;
3 using std::cin;
4 using std::endl;
5
6 int main ()
7 {
8     int i; // loop counter
9     int nN; // to store N
10    int nSum; // to store the sum
11}
```

```

12     cout << "What is N? ";
13     cin >> nN;
14
15     nSum = 0; // initialize
16     for (i=1; i <= nN; i++)
17     {
18         nSum = nSum + i;
19     }
20
21     cout << "The sum of first " << nN << " integers is "
22         << nSum << "." << endl;
23
24     return (0);
25 }
```

Note the initialization that takes place in line 15 and the body of the `for` loop between lines 17 and 19. As we will see next, it is possible to carry out the initialization of the sum in the initialization part of the `for` loop.

Comma Operator

The comma operator `,` is often used in conjunction with some of the statements that we have seen before. The operator is used to separate a list of expressions and returns the value of the last expression that is evaluated. Consider the following statement.

```
nSum = (nX = 2, nY = nX + 3);
```

After the statement is executed, the value of `nX` is 2, the value of `nY` is 5 and finally, the value of `nSum` is 5 since the last expression that is evaluated is `nX + 3`.

An appropriate location to use the comma operator is in the initialization section of the `for` statement. For example, we could rewrite the `for` loop in Example 3.2.3 as follows.

```
for (nSum = 0, i=1; i <= nN; i++)
{
    nSum += i;
}
```

In the initialization part, `nSum` is set to zero and `i` is set to 1. The execution of the `for` loop then begins.

Nested `for` Loops

Finally it is possible to have a nested form of `for` loops. Consider the following statements.

```
int nCount = 0;
for (i=1; i <= n; i++)
{
    for (j=1; j <= m; j++)
    {
        nCount++;
```

```

        }
    }
cout << "Value of nCount is : " << nCount << "\n";

```

The two loops controlled by the loop indices *i* and *j* execute the sole statement *nCount++*; If *n*=10 and *m*=5, then the value of *nCount* after the two loops is 50. One should be careful in making sure that the test condition is not met at some stage of the loop execution; otherwise the loops will be stuck in an infinite loop. Consider the following statements.

```

n=50; fSum = 0.0f;
for (i=1; i <= n; i++)
{
    fSum += static_cast<float>(pow(i,2.0));
    if (i > n/2) i=1; // dangerous
}

```

The execution will be stuck in the **for** loop since the value of *i* will always be less than *n*. Consider the following statements.

```

n=m=10; fSum = 0.0f;
for (i=1; i <= n; i++)
{
    for (j=1; j <= m; i++) // dangerous
    {
        fSum += static_cast<float>(i+j);
    }
}

```

Once again, the execution is stuck in the inner **for** loop since the value of *j* remains at 1!

Tip: It is a common programming error to get stuck in infinite loops because the termination condition is never possible or because of typing the incorrect loop index variable name or because of misplaced semicolons!

3.3 Other Control Statements

There are other mechanisms to control the execution flow in a C++ program.

break syntax

The general syntax of the statement is

```
break;
```

Placing this statement at an appropriate location ensures that the control (or program flow) is immediately shifted to outside the innermost loop that the **break** statement is associated with.

Consider the following example. We wish to compute the sum of the ages of all the students in a class. We will assume that we do not know (or not told) how many students are in the class. The user is

expected to input the ages of the students one at a time. When there are no more students, the user will input the age as zero or a negative number. We will first write the relevant statements using the `do ..while` statement.

```
int nAge;           // to store the student's age
int nSum = 0;       // to store the sum of the ages
do
{
    cout << "Enter the age (zero or negative to end): ";
    cin >> nAge;
    if (nAge > 0)
        nSum += nAge;
} while (nAge > 0);
cout << "The sum of the ages is " << nSum << "." << endl;
```

We will rewrite the program segment using the `break` statement.

```
int nAge;           // to store the student's age
int nSum = 0;       // to store the sum of the ages
for (;;)           // an infinite loop!
{
    cout << "Enter the age (zero or negative to end): ";
    cin >> nAge;
    if (nAge <= 0) break;
    nSum += nAge;
}
cout << "The sum of the ages is " << nSum << "." << endl;
```

Note that the `for` statement with nothing specified for its three components mimics an infinite loop. The `break` statement makes it possible to exit the loop.

continue syntax

The `continue` statement is closely associated with the `break` statement. The general syntax of the statement is

```
continue;
```

Placing this statement at an appropriate location ensures that the statements between the `continue` statement and the end of the nearest loop are not executed.

We will rewrite the example shown with the `break` statement now using the `continue` statement.

```
int nAge;           // to store the student's age
int nSum = 0;       // to store the sum of the ages
do
{
    cout << "Enter the age (zero or negative to end): ";
    cin >> nAge;
    if (nAge <= 0)
        break;
    nSum += nAge;
}
```

```

if (nAge <= 0) continue;
nSum += nAge;
} while (nAge > 0);
cout << "The sum of the ages is " << nSum << "." << endl;

```

switch syntax

One could look at the **switch** statement as a specialized case of **if ... else** statement. The statement provides different execution paths based on the value of an expression that evaluates to an integer. The syntax of the statement is as follows.

```

switch (integral expression)
{
    case ConstantExpression1:
        statement(s)
        break;
    case ConstantExpression2:
        statement(s)
        break;
    ...
    default:
        statement(s)
        break;
}

```

The entire block of statements are enclosed between the braces { ... }. Each of the different execution path begins with the keyword **case** that is followed by an unique constant integer expression and the colon symbol. The **break** keyword at the end of an execution path is optional. If the **break** keyword is used then the subsequent statements are not executed and the control is transferred to the first statement after the switch block. The keyword **default** is optional and the statements in that subblock are executed only if the integral expression is not equal to any one of the constant expressions associated with the different case labels.

A few things to note about the **switch** statement. The switch expression and all the expressions associated with the case labels must be of the same data type. Recall that the primitive data types – **int**, **short**, **long**, and **char**, are all of the integral data type. In addition, these expressions must be constants.

Let us now look at an example. We will write a program segment to compute the grade point average (GPA) of a student's grades in a semester. We will assume that we have a 4-point grading system with all the courses carrying equal credits.

```

char cGrade;          // to store the student's grade
int nCourses = 0;     // to store the total number of courses
float fGPA = 0.0;    // to store the GPA
do
{
    cout << "Enter the Grade (Type S to end): ";
    cin >> cGrade;

```

```

switch (cGrade)
{
    case 'A': nCourses++;
                fGPA += 4.0;
                break;
    case 'B': nCourses++;
                fGPA += 3.0;
                break;
    case 'C': nCourses++;
                fGPA += 2.0;
                break;
    case 'D': nCourses++;
                fGPA += 1.0;
                break;
    case 'E': nCourses++;
                fGPA += 0.0;
                break;
    case 'S':
                break;
    default:
                cout << "Invalid grade.\n";
                break;
}
} while (cGrade != 'S');
if (nCourses != 0) fGPA /= nCourses;
cout << "Student has taken " << nCourses << " and the GPA is "
     << fGPA << "." << endl;

```

goto syntax

The general syntax of the **goto** statement is as follows.

```

goto label;
...
label: statement(s);

```

Programmers have a love-hate relationship with the **goto** statement. Unrestricted use of the **goto** statement can lead to programs that are very difficult to read and maintain. C++ allows the **goto** statement to be used within a function (we will see what functions are in the next chapter). The execution control is transferred from the point where the **goto** is used to the location where the target label is used in the **goto** statement. The label is an (unique) identifier and is followed by a colon. We will now rewrite the program segment used in illustrating the **break** and **continue** statements using the **goto** statement..

```

int nAge;           // to store the student's age
int nSum = 0;       // to store the sum of the ages

begin:
    cout << "Enter the age (zero or negative to end): ";
    cin >> nAge;
    if (nAge > 0)

```

```

{
    nSum += nAge;
    goto begin;
}
cout << "The sum of the ages is " << nSum << "." << endl;

```

Note that the label can appear in any location in the program – before or after it is referenced in any statement in the program.

For the remainder of this chapter we will look at several examples illustrating more selection and repetition concepts, programming styles, algorithm development etc.

Example Program 3.3.1 Using control statements for computing student GPA

The next example completes the problem of computing the student GPA discussed earlier in this chapter.

Program Statement: Develop a program to compute the semester GPA and the cumulative GPA of a student taking courses in a four-point grading system. The program should ask the user to input the (a) current GPA and the number of semester hours taken so far, and (b) the grade and the semester hours for each course taken this semester. It should display as a final report the GPA and the semester hours before the current semester and for the current semester, and the cumulative GPA and semester hours.

Solution: We will first develop the terminology and then the algorithm to solve the given problem. We will use the term **cumulative** to signify the time period prior to the current semester, **semester** to signify the current semester and **new** to signify the state at the end of the current semester. The equation to compute the GPA is

$$GPA = \frac{\sum_{i=1}^n g_i s_i}{\sum_{i=1}^n s_i} \quad (3.3.1)$$

where n is the total number of courses, s_i is the number of semester hours (usually between 1 and 4) for the i^{th} course and g_i is the numerical grade (A=4, B=3, C=2, D=1, E=0) for the i^{th} course. The above equation can also be written as

$$GPA = \frac{\sum_{i=1}^{n_{cum}} g_i s_i + \sum_{j=1}^{n_{sem}} g_j s_j}{\sum_{i=1}^{n_{cum}} s_i + \sum_{j=1}^{n_{sem}} s_j} \quad (3.3.2)$$

where n_{cum} is the (cumulative) number of courses taken so far and n_{sem} is the number of courses taken this semester. Note that

$$\sum_{i=1}^{n_{cum}} g_i s_i = (GPA)_{cum} \sum_{i=1}^{n_{cum}} s_i \quad (3.3.3)$$

where $(GPA)_{cum}$ is the cumulative GPA.

Now to the algorithm.

1. Read the cumulative GPA and semester hours. Are these values valid?
2. Ask the user for the course grade. Loop starts here.
3. If the grade is S exit this loop. Is the grade valid? If yes, ask the user for the number of semester hours for the course. Is the value valid?
4. Track or update $\sum_{i=1}^{n_{sem}} s_i$ and $\sum_{j=1}^{n_{sem}} g_j s_j$.
5. End loop.
6. Use Eqns. (3.3.2) and (3.3.3) to compute the required quantities.
7. Display the results.

The developed program is closely (but not entirely faithful) based on the above algorithm and is shown below.

main.cpp

```

1 #include <iostream>
2
3 int main ()
4 {
5
6     float fNewGPA = 0.0;           // new GPA
7     int    nNewSemHrs = 0;         // new number of sem hours
8     float fCumulativeGPA = 0.0;   // current GPA (excluding this semester)
9     int    nCumulativeSemHrs = 0; // total number of sem hours (excluding
10                                // this semester)
11
12     // get current data on student
13     std::cout << "Enter current GPA: ";
14     std::cin >> fCumulativeGPA;
15     std::cout << "Enter current # of sem. hrs.: ";
16     std::cin >> nCumulativeSemHrs;
17     // TODO: check whether these values are valid.
18
19     // semester-related variables
20     char  cCourseGrade;          // grade for the course
21     int    nCourseSemHrs;         // sem hrs for the course

```

```

22     float fSemGPA = 0.0;           // semester GPA
23     int    nTotalSemHrs = 0;       // total semester hours
24     bool   bError;              // input error indicator
25
26     // loop until done
27     for (;;)
28     {
29         // get course grade
30         std::cout << "Enter the course grade (Type S to end): ";
31         std::cin >> cCourseGrade;
32         if (cCourseGrade == 'S') break;
33
34         // set no error in user input
35         bError = false;
36
37         // get course semester hours
38         std::cout << "Enter course sem hrs: ";
39         std::cin >> nCourseSemHrs;
40         // TODO: check whether this value is valid.
41
42         // branch on the course grade
43         switch (cCourseGrade)
44         {
45             // note the type-cast conversions below
46             case 'A': fSemGPA += 4.0*float(nCourseSemHrs);
47                 break;
48             case 'B': fSemGPA += 3.0*float(nCourseSemHrs);
49                 break;
50             case 'C': fSemGPA += 2.0*(float)nCourseSemHrs;
51                 break;
52             case 'D': fSemGPA += 1.0*static_cast<float>(nCourseSemHrs);
53                 break;
54             case 'E':
55                 break;
56             // invalid grade input
57             default:
58                 std::cout << "Invalid grade " << cCourseGrade << ".\n";
59                 bError = true;
60                 break;
61         }
62         if (!bError)
63             nTotalSemHrs += nCourseSemHrs;
64     }
65
66     // compute (a) new semester hours and GPA
67     //          (b) this semester's GPA
68     nNewSemHrs = nTotalSemHrs + nCumulativeSemHrs;
69     if (nTotalSemHrs > 0)
70         fSemGPA /= float (nTotalSemHrs);      // avoid divide by zero
71     fNewGPA = float(nTotalSemHrs*fSemGPA+nCumulativeSemHrs*fCumulativeGPA)
72                           /float(nNewSemHrs);
73
74     // display the output
75     std::cout << std::endl << " FINAL GRADE REPORT";
76     std::cout << std::endl << " -----" << std::endl;
77     std::cout << "           GPA before this semester: " << fCumulativeGPA
78                           << std::endl;

```

```

79     std::cout << "Total sem. hrs. before this semester: " << nCumulativeSemHrs
80             << std::endl;
81     std::cout << "                                GPA this semester: " << fSemGPA
82             << std::endl;
83     std::cout << "      Total sem. hrs. this semester: " << nTotalSemHrs
84             << std::endl;
85     std::cout << "      Cumulative GPA : " << fNewGPA
86             << std::endl;
87     std::cout << "      Cumulative sem. hrs.: " << nNewSemHrs
88             << std::endl << std::endl;
89 // TODO: display just enough significant digits
90
91 }           // parenthesis is optional
92 }
```

A few things to note about the program. First, the error checks are minimal. The TODO section identifies what needs to be done. This is left as an exercise for the reader. Second, we have not used the `using` keyword with `cout` etc. Instead the `::` (unary scope resolution operator) is used directly with the standard namespace `std`. In other words,

```
#include <iostream>
using std::cout;
...
cout ...
```

is equivalent to

```
#include <iostream>
...
std::cout ...
```

The choice of which style to use in the program is a user decision. Third, the program shows how different data types can be handled appropriately through a process of type casting or explicit data conversion. While GPA can have a fractional component, the number of semester hours is an integer. As we saw in Chapter 2, the result of purely integer arithmetic contains no fractional component – $4/3=1$ as is $5/3=1$. We look at this issue in sufficient detail in the last section the chapter.

We will now look at another example involving several control statements.

Example Program 3.3.2 Using control statements for data analysis

Program Statement: Develop a program to analyze data obtained from a simple Tension Test experiment. The test is carried out to construct what is known as a Load-Deflection diagram. For most materials that are subjected to this test, the initial portion of the graph is linear before the response becomes nonlinear. It is assumed that a maximum to 10 pairs of load-deflection values will be obtained from the test with the starting pair assumed to be $(0,0)$. The load values are assumed to be in the ascending order. The entire graph is to be approximated as piecewise linear segments, i.e. the points on the graph will be connected to each other via straight lines (see Fig. 3.3.1). The slope of the line is to be monitored to see

when (load value) the graph becomes nonlinear. Display a table showing the load-deflection pairs and the slope of each segment. Also display the load value at which the response becomes nonlinear.

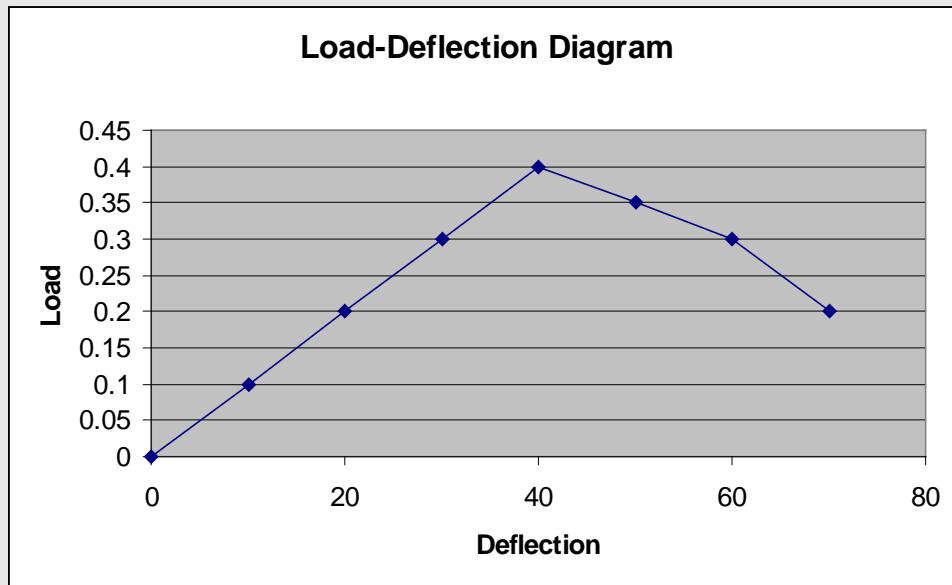


Fig. 3.3.1 A sample load-deflection diagram

Solution: This problem requires that we use the vector data type to store the load values, deflection values and slope values.

The slope, s of a straight line connecting points (x_1, y_1) and (x_2, y_2) is given as

$$s = \frac{y_2 - y_1}{x_2 - x_1} \quad (3.3.4)$$

Now to the algorithm.

1. Initialize all variables.
2. Ask the user for the load and deflection values. The loop starts here.
3. If the load value is zero, exit the loop.
4. Are the values valid? Are the load values in ascending order? Is the load value greater than the previous value?
5. Store the load-deflection pair in the appropriate vectors.

6. Have we reached the limit on the number of pairs of data (10)? If yes, go to step 7. Otherwise go to step 2.
7. Loop through all segments. Number of segments is one less than the number of points.
8. Compute the slope of the line in the segment using Eqn. (3.3.4).
9. End loop.
10. Loop through all segments but the last one.
11. Compare the slope in the current segment with the slope in the next segment. If the slopes are not equal within a specified tolerance, the response has become nonlinear. The load at the end point of the current segment is the load at which the response has become nonlinear.
12. Display the output.

The developed program based on the above algorithm is shown below.

main.cpp

```

1 #include <iostream>
2 #include <iomanip>
3 #include <ios>
4 #include <cmath>
5 #include <string>
6
7 int main ()
8 {
9     const int MAXVALUES = 11;      // maximum # of storage locations
10    // for load and deflection values
11    double dVLoadData[MAXVALUES];
12    double dVDeflectionData[MAXVALUES];
13    double dVSlopes[MAXVALUES];
14    std::string szVPrompts[] = {"first", "second", "third", "fourth",
15                                "fifth", "sixth", "seventh", "eighth",
16                                "ninth", "tenth"};
17
18    // obtain user input
19    double dLoad;
20    double dDeflection;
21
22    // initialize
23    int nPoint=0;                  // number of data points
24    dVLoadData[nPoint] = 0.0;      // first point at (0,0)
25    dVDeflectionData[nPoint++] = 0.0;
26
27    // loop until no more input
28    for (;;)
29    {
30        std::cout << "Input " << szVPrompts[nPoint-1] <<
31                    " load and deflection values (0 load to end): ";
32        std::cin >> dLoad >> dDeflection;

```

```

33         // zero load signifies end of input
34     if (dLoad == 0.0)
35         break;
36     // load or deflection cannot be negative or zero
37     else if (dLoad < 0.0 || dDeflection <= 0.0)
38         std::cout << "Load and deflection must be positive.\n";
39     // ordered input?
40     else if (dLoad <= dVLoadData[nPoint-1])
41         std::cout << "Load values must be in ascending order.\n";
42     else
43     {
44         // save the values and update counter
45         dVLoadData[nPoint] = dLoad;
46         dVDeflectionData[nPoint++] = dDeflection;
47
48         // no more storage space?
49         if (nPoint == (MAXVALUES-1))
50             break;
51     }
52 }
53
54
55 // compute slope of each segment
56 int i;                                // loop index
57 int nSegments = nPoint-1;
58 for (i=0; i < nSegments; i++)
59 {
60     dvSlopes[i] = (dVLoadData[i+1] - dVLoadData[i]) /
61                     (dVDeflectionData[i+1] - dVDeflectionData[i]);
62 }
63
64 // find first load level at which response is nonlinear
65 double dTolerance = 1.0e-3;           // tolerance
66 double dNonlinLoadValue = 0.0;        // (nonlinear) load level
67 for (i=0; i < nSegments-1; i++)
68 {
69     if (fabs(dvSlopes[i+1] - dvSlopes[i]) > dTolerance)
70     {
71         dNonlinLoadValue = dVLoadData[i+1];
72         break;
73     }
74 }
75
76 // output the results
77 std::cout << std::endl << " FINAL REPORT";
78 std::cout << std::endl << " ----- " << std::endl;
79 std::cout << "Load " << " " << "Deflection" << " "
80             << "Slope\n";
81 std::cout << "---- " << " " << "-----" << " "
82             << "----\n";
83 std::cout << std::setiosflags(std::ios::left)
84             << std::setw(7) << dVLoadData[0] << " "
85             << std::setw(10) << dVDeflectionData[0]
86             << std::endl;
87
88 for (i=1; i < nPoint; i++)
89 {

```

```

90         std::cout << std::setiosflags(std::ios::left)
91             << std::setw(7) << dvLoadData[i] << " "
92             << std::setw(10) << dvDeflectionData[i] << " "
93             << std::setw(10) << dVSlopes[i-1] << std::endl;
94     }
95
96     if (dNonlinLoadValue == 0.0)
97         std::cout << "\nResponse is entirely linear.\n";
98     else
99         std::cout << "\nResponse becomes nonlinear at " <<
100            dNonlinLoadValue << std::endl;
101
102    return 0;
103 }
```

Once again a few things to note about the program.

- (1) First is the definition and usage of vector data types. The primitive data types – `int`, `float`, `double` and `char` are easy to define and use.

```
datatype variablename[const int value];
```

Since character strings is not a primitive data type, we use the standard `string` class when it is required to define and use a vector of character strings. A string variable can store a character string of unknown or variable length. For example,

```
string szName;
string szDate ("Dec 12, 2001");
```

are example declarations of string variables. `szName` initially contains no characters whereas `szDate` contains the string (including blanks or spaces) defined between the “ ” symbols. If later in the program, the name is to be used or defined, one can use `szName` in several different ways as shown below.

```
szName = "John Doe";
szName = " John " + " Doe ";
std::cin >> szName;
std::cout << "Name is: " << szName;
```

The above statements show that the standard `string` variable behaves very similar to the primitive data type. We will learn more about the standard `string` class throughout the text. Once again, note that when the vector variable is defined to contain **n** elements, the indexing to access these elements starts at **0** and ends at **n-1**. For example, with the following declarations

```
string szVDays[7];
float fVX[3];
```

it would be illegal (the program will compile without any error messages!) to have the following statements

```
szVDays[ 7 ] = "Saturday";
fVX[ 3 ] = 10.45;
```

The results or behavior are unpredictable if these statements are executed in the program. When vectors are initialized as in line 14, it is not necessary to explicitly state the size of the vector. The compiler is able to figure out the required size.

(2) MAXVALUES is defined as 11 not 10 since the first pair of values is assumed to be (0,0) and will be stored at [0] locations of the load and deflection data vectors.

(3) The nPoint variable is used to keep track of the current point on the load-deflection diagram. It is initialized to zero in line 23, and the load and deflection are initialized at the first location to zero in lines 24 and 25. Note how nPoint value is updated as nPoint++ (nPoint becomes 1) in line 25 so that the load value can be correctly stored in line 46. This strategy is used once again in line 47.

(4) Logical expressions are used to check the validity of the user input in line 38 and line 41.

(5) The break statement is used to exit the loop in line 36 if the user input for the load value is 0 and in line 51 if ten values are defined by the user.

(6) Eqn. (3.3.4) is implemented in lines 60 and 61. A safety check is not implemented to avoid a divide by zero error. This can be easily done and is left as an exercise.

(7) Step 11 of the algorithm is implemented in the loop in lines 67 through 74. A tolerance value of 10^{-3} is used in checking whether slopes of adjacent segments are equal or not. The use of fabs math function requires the use of cmath header file. Once again, the break statement (line 72) is used to exit the loop.

(8) Finally, let's look at the statements to format and display a table – a number of columns with a header and a fixed (column) width where data are displayed in each row. The three column headers (“Load”, “Deflection” and “Slope”) are defined in lines 79 through 82. We will left-justify the entries in each column via the `std::setiosflag(std::ios::left)` statement. Next we will assume that a field width of 7 is adequate to represent the load values. The column width is chosen as the larger of two numbers – the number of digits required to represent the value and the number of characters in the column header. The `<<` operator formats the floating value so that the display is as compact and efficient as possible. Hence with a field width of 7 we should be able to display (positive) values between 0 and 999999 including values as 0.15, 0.16667, 3000.12 etc. A field width of 10 is used with the deflection values since the column header Deflection contains 10 characters. The same logic applies in formatting the last column containing slope values. We will see more about formatted output in the last section of this chapter.

The following example is a precursor to programs involving numerical analysis and solution techniques.

Example Program 3.3.3 Exhaustive search or trial-and-error

Program Statement: The coefficient of restitution, c is a measure of the elasticity of the collision between two objects one of which is usually at rest. For example, if a ball is dropped from a height H onto a floor and is observed to bounce to a height h , the coefficient of restitution can be computed as

$$c = \sqrt{\frac{h}{H}} \quad (3.3.5)$$

Clearly if conservation of energy principle is followed $0 \leq c \leq 1.0$.

A ball is dropped from a height of 3.5 m onto a floor. The coefficient of restitution between the ball and floor is 0.9. Compute how many bounces occur before the ball bounces to a height as close to 1 m as possible.

Solution: Numerical solution is an attractive approach if the analytical solution is difficult to compute. However, in this problem, even though we know the analytical solution, we will use trial-and-error approach (or exhaustive search) to find the solution. The motivation is to gain confidence in the development of the trial-and-error approach by comparing the trial-and-error solution to the analytical solution.

The basic idea is to increment i , compute the new height h_i and compare the new height to the target height, h_{target} that is 1 m. We will keep track of the difference between the computed height and the target height

$$h_{diff} = |h_i - h_{target}|$$

The difference should decrease with increasing i , and then start increasing. Only under certain set of values will this difference be zero or nearly zero – it is unlikely that the bounced height will be exactly the target height.

The analytical solution to this problem can be found as follows. Let the height to which the ball bounces after every subsequent bounce be denoted as h_1, h_2, h_3, \dots . Note that

$$h_1 = c^2 H \quad h_2 = c^2 h_1 = c^4 H \quad h_i = c^{2i} H \quad (3.3.5a)$$

from which one could solve for i as

$$(2i)\log(c) = \log\left(\frac{h_i}{H}\right)$$

$$\text{or, } i = \frac{\log(h_i/H)}{2\log(c)} \quad (3.3.6)$$

Algorithm: Here is the developed algorithm. As an added safety, we keep track of the number of iterations (or bounces). If the number exceeds a predefined maximum number, we exit the loop. This is one more way we can avoid getting stuck in an infinite loop.

1. Obtain user input for initial height, coefficient of restitution and the target height. Set $(h_i)_{new} = (h_i)_{old} = H$, $(h_{diff})_{new} = (h_{diff})_{old} = H$, and $i = 0$.
2. Loop to compute the new height.
3. Increment i . If $i > Max\ iterations$, exit the loop and print an error message.
4. Compute $(h_i)_{new} = c^2 (h_i)_{new}$, and $(h_{diff})_{new} = |(h_i)_{new} - h_{target}|$.
5. If $(h_{diff})_{new} > (h_{diff})_{old}$ we have found the solution. Set $(h_i)_{new} = (h_i)_{old}$, $i = i - 1$ and exit the loop.
6. Set $(h_{diff})_{old} = (h_{diff})_{new}$ and $(h_i)_{old} = (h_i)_{new}$.
7. Go to step 2.
8. Print the results. The number of bounces is i and the bounced height closest to the target height is $(h_i)_{new}$.

main.cpp

```

1  #include <iostream>
2  #include <cmath>
3
4  int main ()
5  {
6      float fInitialHeight;    // height from which ball is dropped
7      float fCoefofR;          // coefficient of restitution
8      float fTargetHeight;    // target height
9      int nCurIter = 0;        // current iteration
10     bool bError = false;    // error indicator
11     float fHeightNew;
12     float fHeightOld;
13     float fDiffHeightOld;
14     float fDiffHeightNew;
15     const int MAXITERATIONS = 1000;
16
17     // get the input from the user
18     std::cout << "Initial height: ";
19     std::cin >> fInitialHeight;
20     std::cout << "Coefficient of Restitution: ";
21     std::cin >> fCoefofR;
22     std::cout << "Target height: ";
23     std::cin >> fTargetHeight;

```

```

24
25      // initialize
26      fHeightNew = fHeightOld = fInitialHeight;
27      fDiffHeightOld = fInitialHeight;
28
29      // loop until convergence
30      for (;;)
31      {
32          // increment iteration counter
33          nCurIter++;
34
35          // too many iterations?
36          if (nCurIter > MAXITERATIONS)
37          {
38              bError = true;
39              break;
40          }
41
42          // compute new height
43          fHeightNew *= static_cast<float>(pow(fCoefofR, 2.0));
44
45          // compute difference in heights
46          float fD = fHeightNew - fTargetHeight;
47          fDiffHeightNew = static_cast<float>(fabs(fD));
48
49          // passed the lowest difference point?
50          if (fDiffHeightNew > fDiffHeightOld)
51          {
52              // yes. this is the solution.
53              fHeightNew = fHeightOld;
54              nCurIter--;
55              break;
56          }
57
58          // update solution
59          fDiffHeightOld = fDiffHeightNew;
60          fHeightOld = fHeightNew;
61      }
62
63      // analytical prediction
64      float fDiffL, fDiffU;
65      float fBounces = log(fTargetHeight/fInitialHeight) /
66          (2.0*log(fCoefofR));
67      int nUpper = static_cast<int>(ceil(fBounces));
68      int nLower = (nUpper <= 2 ? 1 : (nUpper - 1));
69      fDiffL = static_cast<float>(pow(fCoefofR, 2.0*nLower)*fInitialHeight);
70      fDiffL = static_cast<float>(fabs(fDiffL - fTargetHeight));
71      fDiffU = static_cast<float>(pow(fCoefofR, 2.0*nUpper)*fInitialHeight);
72      fDiffU = static_cast<float>(fabs(fDiffU - fTargetHeight));
73      int nBounces = (fDiffL < fDiffU? nLower : nUpper);
74      float fHeight = static_cast<float>
75          (pow(fCoefofR, 2.0*nBounces)*fInitialHeight);
76
77      // print result
78      std::cout << std::endl;
79      if (bError)
80          std::cout << "Unable to find the solution.\n";

```

```

81     else
82         std::cout << "Numerical Solution" << "\n"
83             << "    Initial height: " << fInitialHeight << "\n"
84             << "    Target height: " << fTargetHeight << "\n"
85             << "Coef of Restitution: " << fCoefofR << "\n"
86             << "    Number of bounces: " << nCurIter << "\n"
87             << "    Height reached: " << fHeightNew << "\n\n"
88             << "Analytical solution" << "\n"
89             << "    Number of bounces: " << nBounces << "\n"
90             << "    Height reached: " << fHeight << "\n";
91
92     return 0;
93 }
```

The program follows the algorithm except for statements 63 through 75 where we compute the analytical solution. Eqn. (3.3.6) is implemented in lines 65-66. However, since the number of bounces is an integer, we use the `ceil` function to find the next higher integer (`nUpper`). In order to find the point that is the closest to the target height, we also use the next lowest bounce (`nLower`). We use these two integer values to compute the height as per Eqn. (3.3.5a), and then find the bounce that is closest to the target height.

3.4 Tying Loose Ends

Before we finish the chapter, we will look at a few of C++ features that will enable us to write better programs.

enumerated types

C++ provides for a user-defined type called enumeration. The general syntax is as follows.

```
enum enumeration_type {list of integer constants};
```

Let us consider the example of the different types of polygons. We could define the enumeration type as follows.

```
enum Polygons {TRIANGLE=1, QUADRILATERAL, PENTAGON, HEXAGON, HEPTAGON,
OCTAGON, NONAGON, DECAGON};
```

`Polygons` now is treated as a data type just as `int`, `float` etc. The compiler assigns integer values such that `TRIANGLE` has a value of 1, `QUADRILATERAL` has a value of 2, `PENTAGON` has a value 3, and so on. Note that by default, if `TRIANGLE` is not assigned a value, it is taken as a zero.

Here is an example usage of this enumeration type in a program following the above definition.

```
Polygons PolyType;           // variable to store the polygon type
std::string szPolyType;

// get the polygon type from the user
std::cout << "Polygon shape (lower case)? ";
std::cin << szPolyType;

if (szPolyType == "triangle")
```

```

PolyType = TRIANGLE;
else if (szPolyType == "square" || szPolyType == "rectangle" )
    PolyType = QUADRILATERAL;
else
    std::cout << "Unsupported polygon type.\n";

```

Because of the usage of the enumeration type, the program is more readable than using simple integer constants to differentiate the types of polygons.

Type Coercion and Casting

How do we handle expressions where different data types are involved? In line 46, the variable fSemGPA on the left side of the assignment (symbol) is a floating value. On the right side of the assignment, we have a floating point constant, 4.0, and an integer variable, nCourseSemHrs. In order to ensure that the computations are done correctly so that fractional components are preserved, we need to convert (or promote) the integer variable nCourseSemHrs. This is done three different ways. The keyword float can be used as a qualifier as

```

float (expression)
or      (float) expression

```

The preferred style is to use the unary cast operator

```
static_cast<float>(expression)
```

The value of the expression in each case is stored temporarily as a floating point number that is then used in the subsequent computations. Note that we could have written statement 46 as

```
fSemGPA += static_cast<float>(4*nCourseSemHrs);
```

Here are some rules governing type coercion when dealing with arithmetic and relational expressions and assignment operations. Note that promotion or widening of a data type involves conversion of a value from a “lower” type to a “higher” type according to a programming language’s precedence of data types.

- (1) Step 1: Each `char`, `short`, `bool`, or enumeration value is promoted to `int`. If both operands are now `int`, the result is an `int` expression.
- (2) Step 2: If Step 1 still leaves a mixed type expression, the following precedence of types is used:

Lowest → highest
int, unsigned int, long, unsigned long, float, double, long double

The value of the operand of the lower type is promoted to that of the higher type, and the result is an expression of that type.

Consider the following expression (`nValue + 4.0`) where `nValue` is defined as an `int`. Step 1 indicates that this expression is a mixed expression. Hence, `nValue` is temporarily coerced to a `double`, and the entire expression is evaluated as a `double`.

Similarly, in relational expressions of the form `nValue1 > fValue1`, the value of `nValue1` is temporarily coerced to a `float` before the comparison takes place.

Formatted Output

C++ provides the programmer with sufficient controls to tailor or format the output to a file or the screen. In Section 2.5 we looked at the very basics of formatted output using `precision`, `setprecision`, `width` and `setw` commands. In this section, we will see a few more ways of controlling the output format. Every output stream has a member function called `setf` that can be used to specify the type of desired output. In conjunction with the member constants in the `ios` class, various operations can be carried out. Note that the `ios` class is described (hence included) in the `<iostream>` and `<fstream>` header files.

Table 3.4.1 Formatting with `setf`

Flag	Effects of setting the flag	Default
<code>ios::showpos</code>	Shows a plus sign before positive integers.	Not active
<code>ios::showpoint</code>	For floating point numbers, the decimal point and the trailing zeros are shown. If this flag is not set, a floating point number without a fractional component may not have the decimal point and the trailing zeros displayed.	Not active
<code>ios::fixed</code>	The scientific notation (using <code>e</code>) is not used. Instead the floating-point numbers are output completely. This unsets the <code>ios::scientific</code> flag.	Not active
<code>ios::uppercase</code>	An uppercase <code>E</code> is used instead of the lowercase <code>e</code> in scientific notation.	Not active
<code>ios::scientific</code>	Floating point numbers are written in scientific notation using the <code>e</code> symbol.	Not active
<code>ios::right</code>	If the field width is specified (using the <code>width</code> function or <code>setw</code> manipulator), the next item output is right justified.	Default
<code>ios::left</code>	If the field width is specified (using the <code>width</code> function or <code>setw</code> manipulator), the next item output is left justified.	Not active
<code>ios::resetiosflags</code>	Clears the flag so that a new setting can be specified. For example, if the current output is left justified, then to have right justification be applicable,	use

```
resetiosflags(ios::adjustfield)before      using
ios::right.
```

The flags invoked using the `setf` function can be removed using the `unsetf` function as we will see in the following example.

Example Program 3.4.1 C++ Stream Input/Output

Program Statement: Develop a simple program to understand how formatting statements works with floating point numbers.

Solution: We will develop an interactive program in which the user will be asked to input a floating-point number and the output precision. The number will be read in and then displayed on the screen in various output styles. The program will run in a continuous loop with the program termination possible by typing in CTRL-C.

The resulting program is shown below.

main.cpp

```

1   #include <iostream>      // also contains ios definitions
2
3   int main ()
4   {
5       float fX;          // to store the user input
6       int nPrecision; // to store the user-specified precision
7
8       for (;;)
9       {
10           // get the number from the user
11           std::cout << "Input a floating point number plus precision: ";
12           std::cin >> fX >> nPrecision;
13
14           // display values in default settings
15           std::cout << "You have input (default): " << fX << '\n';
16
17           // set output style 1 (precision, not e-format, show decimal
18           // point and trailing zeros)
19           std::cout.precision (nPrecision);
20           std::cout.setf (std::ios::fixed | std::ios::showpoint);
21           std::cout << "You have input (format style 1): " << fX << '\n';
22
23           // modify - style 2 (in addition, show plus sign, if applicable)
24           std::cout.setf (std::ios::showpos);
25           std::cout << "You have input (format style 2): " << fX << '\n';
26
27           // modify - style 3 (invoke scientific format)
28           std::cout.unsetf (std::ios::fixed);
29           std::cout.setf (std::ios::scientific);
30           std::cout << "You have input (format style 3): " << fX << '\n';
31
32           // restore default settings
33           std::cout.unsetf (std::ios::showpoint)
```

```

34                     | std::ios::showpos | std::ios::scientific);
35             std::cout.setf (0, std::ios::floatfield);
36         }
37
38     return 0;
39 }
```

The program enters an infinite loop in line 8. The user input is obtained in line 12. The input number is displayed in four different styles. First, the default style is displayed. This style is compiler dependent. We use the user-specified precision to control how many digits after the decimal point are displayed. In the first user-defined style, we set the style as not showing the value in the exponent (scientific) form and always showing the decimal point as well as the trailing zeros. This is carried out in line 20. Using the bitwise `|` operator, we are able to carry out in one statement what can also be done in two statements. In other words, line 20 is equivalent to the following two statements

```

std::cout.setf (std::ios::fixed);
std::cout.setf (std::ios::showpoint);
```

The bitwise operator can be used to stack as many different types of output specifications as required. Once a style (or flag) is set, it remains in effect until it is reset or unset. In the second user-defined style, in addition to the existing style, we specify that the `+` or the `-` sign be displayed before the number. This is specified in line 24. Finally, in the third user-defined style, we want to display the numbers in the scientific notation (line 29). Line 28 is necessary to unset the current style – fixed or non-exponent way of displaying float values. Finally, the default settings are restored in lines 33-35 by unsetting the existing flags, and restoring the default floating point settings in line 35.

A sample output from the program is shown in Fig. 3.4.1.

```

"c:\data\Winword\Books\Oop\programs\Example3_4_1\Debug\Example3_4_1.exe"
Input a floating point number plus precision: 3.0956 5
You have input (default): 3.0956
You have input (format style 1): 3.09560
You have input (format style 2): +3.09560
You have input (format style 3): +3.09560e+000
Input a floating point number plus precision: -0.0073096789 5
You have input (default): -0.0073097
You have input (format style 1): -0.00731
You have input (format style 2): -0.00731
You have input (format style 3): -7.30968e-003
Input a floating point number plus precision: _
```

Fig. 3.4.1 Sample output (MS .Net 2003)

Example Program 3.4.2 Creating a Formatted Table

Program Statement: Obtain not more than 10 floating point numbers from the user. Display the numbers on the screen as a two column table – the first column should show the index and the second column the value of the floating point numbers.

Solution: The program development is quite straightforward. We will define a float vector of size 10. The user will be prompted to input the number of input values. This will be followed by the user inputting one value at a time. Once all the values are read, we will output the table as per the specifications – two columns with the first column showing the index and the second column showing the corresponding value.

The resulting program is shown below.

main.cpp

```

1  #include <iostream>           // also contains ios definitions
2  #include <iomanip>          // to define setw manipulator
3
4  int main ()
5  {
6      const int MAXVALUES = 10;
7      float fVX[MAXVALUES];      // to store the user input
8      int nUserValues;          // # of user-defined values
9      int i;                    // loop index
10
11     // how many values to store and display?
12     do
13     {
14         std::cout << "How many values are there? ";
15         std::cin >> nUserValues;
16     } while (nUserValues <= 0 || nUserValues > MAXVALUES);
17
18     // get the values from the user
19     for (i=0; i < nUserValues; i++)
20     {
21         std::cout << "Input value " << i+1 << ": ";
22         std::cin >> fVX[i];
23     }
24
25     // set output style
26     std::cout.setf (std::ios::scientific);
27     // table column headings
28     std::cout << "    " << "Index" << "        " << "Value" << '\n';
29     std::cout << "    " << "----" << "        " << "----" << '\n';
30     for (i=0; i < nUserValues; i++)
31     {
32         std::cout << "    " << std::setw(5) << i+1;
33         std::cout.setf (std::ios::showpos);
34         std::cout << "    " << std::setw(10) << fVX[i] << '\n';
35         std::cout.unsetf (std::ios::showpos);
36     }
37
38     return 0;

```

39 }

Lines 12-16 show the code to obtain a valid value for the number of user-defined values. These values are then obtained and stored in lines 19-23. To display the float values in the scientific style, the `setf` function is used in line 16. Next the column headings are displayed in lines 28 and 29. The number of blank spaces used before displaying both the index and the value is a function of the field width used to display both these values. The field width is set to 5 for displaying the index. Unlike other formatting flags, the field width manipulator, `setw` is valid only for the next item that follows the specification. This is set in line 32. Similarly, in line 34, the field width is set to 10 display the value. Is this field width adequate to display any floating point value?

Summary

In this chapter we saw most of the important C++ control structures. At this stage with the knowledge gathered from Chapters 2 and 3, we should be able to write moderately complex programs.

Below we summarize the important facts learnt in this chapter.

- Definitions and declarations must precede usage. Look at C++ compiler having access to a C++ dictionary and a thesaurus that is augmented by a user-defined dictionary and thesaurus. The compiler looks first at these documents when it is compiling a program. If it cannot find a declaration or a definition it issues an error message.
- Selection is achieved through (a) `if ... else if ... else` statement, and (b) `switch` statement. The selection conditions can be formed through expressions involving relational and logical operators.
- Repetition is achieved through (a) `while` statement, (b) `do ... while` statement, and (c) `for` statement.
- In addition the `break`, `continue` and `goto` provide other forms of control statements. Later in the book we will see the `return` and the `exit` statements.
- At the end of the chapter, we saw (a) the enumerated type that help in making programs easier to read, and (b) the dangers in mixed type arithmetic (type coercion).
- C++ provides a rich set of formatting controls for both screen as well as file outputs. We looked at a few of them in this chapter. In later chapters we will see the rest.

EXERCISES

When writing programs simple or complex, it is a good programming habit to check for errors in input and inform via clear error messages as to why the input has an invalid value. When a problem specifies exhaustive search (or trial and error), assume that you do not know the analytical solution. You may use the analytical solution to check the computer program generated solution.

Appetizers

Problem 3.1

Write a program to display as a table the values of the function $y(x) = ax^3 + bx^2 + cx + d$ for the range $-10 \leq x \leq 15$ using an increment supplied by the user. Obtain the values of the coefficients of the cubic polynomial and the increment from the user.

Problem 3.2

Write a program to obtain a set of up to a maximum of 10 positive float numbers. For this set of numbers, compute its (a) average, (b) minimum, (c) maximum, and (d) standard deviation. Prompt the user to enter the numbers one at a time. A negative value signals the end of the user input. Check for some basic input errors such as the first number being negative etc.

Problem 3.3

Write a program to obtain a set of (x, y) coordinates for 5 points. Find the two points that are (a) closest to each other, and (b) farthest apart from each other.

Problem 3.4

Write a program to accept (a) an integer input and print out the number with the digits reversed (for example, if the input is -18080, the output should be 08081-), and (b) a string input and print out the string with the characters reversed (for example, if the input is arizona, the output is anozira).

Problem 3.5

What is the output from the following statements?

```
int nV=0;
for (int i=1; i <= 50; i = 3*i)
    nV++;
cout << "nV is " << nV << "." << endl;
```

Main Course

Problem 3.6

Write a program to compute and display the value of sine of an angle using the following formula.

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots + \frac{(-1)^{i+1} x^{2i-1}}{(2i-1)!} + \dots$$

Terminate the series if the difference between two consecutive numbers in the series is less than 10^{-4} . Compare your results with the value provided by the `sin` function.

Problem 3.7

Rewrite Example Program 3.2.4 with the following changes. The user input should be as follows - (a) the initial input should be the student last name, first name, current GPA and the current number of semester hours, and (b) for the current semester, the input should be the course number, the course semester hours, and the raw score on the course (between 0 and 100). No more course input is required if the course number is STOP. The raw score should be translated to a letter grade as follows – A : 91-100, B : 81-90, C : 71-80, D : 61-70, and E : 0-60. Assume that a student can take at most 6 courses per semester. Print a Grade Report showing the student's name, details of the current semester, and the new (cumulative) semester hours and GPA.

Problem 3.8

The differential equation for a transverse deflection of a beam is given by $\frac{d^4 y}{dx^4} = \frac{w(x)}{EI}$. The solution for a simply-supported beam of length L subjected to a uniform loading, w is given as

$$y(x) = -\frac{wx}{24EI} (x^3 - 2Lx^2 + L^3)$$

Write a program to compute the largest deflection and its location using trial and error. Obtain the values of L, E, I, w and the units for force and length from the user. Assume that the user input is in consistent units and no unit conversion is to be carried out.

Problem 3.9

What is the output from the following statements?

```
int nV=0;
for (int i=1; i <= 50; i = 3*i);
    nV++;
cout << "nV is " << nV << "." << endl;
```

Problem 3.10

What is the output from the following statements?

```
int nV=0;
```

```

for (int i=0; i <= 50; i = 3*i);
    nV++;
cout << "nV is " << nV << "." << endl;

```

C++ Concepts

Problem 3.11

Newton's Law of Cooling is given as $\frac{dT}{dt} = -k(T - T_{\infty})$ where T is the temperature that is a function of time, k is a positive constant, T_0 is the temperature at time $t = t_0$, and T_{∞} is the ambient temperature. Solution of the above equation is given as

$$T(t) = T_{\infty} + (T_0 - T_{\infty})e^{-kt}$$

A coroner is called to a crime site (a warehouse) at 1 am on Jan 15, 2009. She finds a corpse whose temperature then is 80°F . The temperature in the warehouse is maintained at 62°F . After two hours the temperature of the corpse drops to 72°F . Write a program that uses trial and error to find the date and time of death. (*Hint:* The coroner when questioned says that for most situations $0.1/\text{hr} \leq k \leq 0.5/\text{hr}$.)

Problem 3.12

A cable suspended between two posts hangs in the form of a catenary whose equation is given as

$$y(x) = \frac{T}{w} \cosh\left(\frac{w}{T}x\right) + c$$

where $y(x)$ is the height of the cable above the ground, T is the tension in the cable, w is the weight of the cable per unit length, and c is a constant. By measuring the height of the cable, the following conditions are known - $y(x = -100) = 750$, $y(x = 0) = 75$ and $y(x = 100) = 750$. Find the values of the parameters in the catenary equation so as to satisfy the given conditions as closely as possible using trial and error.

Problem 3.13

Write a program to obtain 7 values from the user. Store and display (a) these seven original values, (b) the seven sorted values, and (c) their mean, median and standard deviation. Your approach should be general enough so that it is applicable for any set of numbers. (*Hint:* Use Selection Sort. The basic idea is to determine the minimum (or maximum) of the list and swap it with the element at the index where it is supposed to be. The process is repeated such that the n^{th} minimum (or maximum) element is swapped with the element at the $(n-1)^{\text{th}}$ index of the list. Here is an example involving 8 integer numbers and sorting in ascending order.)

Initial	8	6	10	3	1	2	5	4
Pass 1	1	6	10	3	8	2	5	4
Pass 2	1	2	10	3	8	6	5	4
Pass 3	1	2	3	10	8	6	5	4
Pass 4	1	2	3	4	8	6	5	10
Pass 5	1	2	3	4	5	6	8	10
Pass 6	1	2	3	4	5	6	8	10
Pass 7	1	2	3	4	5	6	8	10

Chapter 4

Modular Program Development

“Ambition is a lust that is never quenched, but grows more inflamed and madder by enjoyment.”

Thomas Otway

“Build a better mousetrap and the world will beat a path to your door.” Ralph Waldo

Emerson

“It is not the greatness of a man's means that makes him independent, so much as the smallness of his wants.” William Cobbett

We saw a number of C++ constructs in Chapters 2 and 3. Using these constructs we can write moderately complex programs. However, as the complexity of the tasks increases, developing a single long program contained entirely in the main program is neither efficient nor practical. In this chapter we will learn about the elements of modular program development starting with functions, scope of variables and finally, program development with multiple source files.

Objectives

- To understand and practice the concept of functions.
- To understand the concept of scope of variables.
- To understand the concept of modular program development.
- To understand and practice good programming styles.

4.1 Functions

We have in the previous chapters used functions without being formally introduced to them. In Example 2.5.2, we used the `sin` function to compute the sine of an angle. As the documentation shows in Fig. 2.7.5, the function expects as an argument the angle in radians as a double precision value and returns the sine of the angle as a double precision value. One can look at a function as a component in a program (an independent unit that can be separately compiled) that is defined because either it has a very specific functionality or because this functionality is used in several locations in the program or both. Let us look at the `sin` example again.

```
dAngle = 0.2;
double dValue = sin(dAngle);
```

The `sin` function expects to see a single parameter – a double precision value. In the above example, the variable `dAngle` provides that value in radians. The `sin` function uses that value as the angle and evaluates the sine of the angle. The computed value is returned to the calling program. The returned value can be stored and used via a variable. As shown above, the returned value is stored in the variable `dValue`.

Functions can be passed no arguments or one or more arguments, and can return either nothing or a single value. The general syntax is as follows.

```
returnvalue functionname (argument 1, argument 2, ...)
// argument list is optional
{
...
    return somevalue; // optional. required only if
                      // returnvalue is not void.
}
```

In fact the `main` program is a special function whose return value is an `int` and has a variable number of function parameters. The `functionname` just like variable names must be unique. The only exception is when a function is overloaded. We will see overloaded functions in Section 4.2. Here are some examples of functions.

Example 1

```
int IntSquare (int n)    // computes the square of an integer number
{
    int nValue = n*n;
    return nValue;
}
```

Example 2

```
float SumSquares (float x, float y) // computes the sum of the squares
{
    return (x*x + y*y);
}
```

Example 3

```

bool IsEven (int n)      // determines if a number is even or not
{
    if ((n % 2) == 0)
        return true;      // even number
    else
        return false;     // odd number
}

```

In the three examples, the number of function parameters is either one or two, and one value is returned to the calling program. The variable name in the argument list as used in the calling program need not be the same as the corresponding variable name as used in the function (see example below where the calling program uses `nNumber` and the function uses `n`). Note that each of the function definitions states that the argument must evaluate to an integer (`IntSquare` and `IsEven`) or a float (`SumSquares`). For example, in the case of an integer argument, in the calling program the argument can be an integer constant, an integer variable or an expression that evaluates to an integer. In the `IsEven` function we have two return statements. Sometimes the `return` statement returns no value but merely terminates the execution of a function and returns control to the calling program. Before functions are used anywhere in a program, they must be declared.

Example Program 4.1.1 Function to determine if a number is even or odd

Let us see how one would use the `IsEven` function in a program. The function can be developed quite easily if one recognizes that an even number is exactly divisible by 2. In other words, the remainder is zero when an even number is divided by 2.

main.cpp

```

1  #include <iostream>
2  bool IsEven (int);      // function prototype
3
4  int main ()
5  {
6      int nNumber;      // to store user input
7
8      std::cout << "Type an integer value: ";
9      std::cin >> nNumber;
10
11     if (IsEven(nNumber))
12         std::cout << nNumber << " is an even number.\n";
13     else
14         std::cout << nNumber << " is an odd number.\n";
15
16     return 0;
17 }
18
19 bool IsEven (int n)      // determines if n is even or not
20 {
21     if ((n % 2) == 0)    // compute remainder
22         return true;    // even number

```

```

23     else
24         return false; // odd number
25 }

```

The function `IsEven` is declared in line 2 before being used in line 11. This is called a function prototype. The prototype establishes the return data type, the number of parameters and the data type associated with each parameter. The names of the variables used in the parameter list can be provided but are not necessary. In other words, one could have written the prototype as

```
bool IsEven (int n);
```

Since the function needs to be declared before being used, we could have written the program differently as follows.

```

#include <iostream>

bool IsEven (int n)      // determines if a number is even or not
{
    if ((n % 2) == 0)
        return true; // even number
    else
        return false; // odd number
}

int main ()
{
    int nNumber; // to store user input

    std::cout << "Type an integer value: ";
    std::cin >> nNumber;

    if (IsEven(nNumber))
        std::cout << nNumber << " is an even number.\n";
    else
        std::cout << nNumber << " is an odd number.\n";

    return 0;
}

```

In this version, the function `IsEven` is defined at the top of the file before the `main` program. Hence the function prototype is not necessary. This solution does not work if the source program exists in several files. As we will see in the next section, prototyping is a generic solution when functions are used.

How do we define functions where there is no need to return a value or if there are no parameters to be passed? C++ has a keyword called `void` that can be used. Here is an example of a function that neither returns a value nor has a function parameter.

Example 4

```

void PrintInputError (void)
{
    std::cout << "Your input is invalid.\n";

```

```
}
```

OR

```
void PrintInputError ()
{
    std::cout << "Your input is invalid.\n";
}
```

Example 5

```
void DisplayCoordinates (float x, float y)
{
    cout << "X Coordinate: " << x << ". Y Coordinate: " << y << '\n';
}
```

To correctly use a function with multiple parameters, one must be careful in ordering the arguments. Consider the following example that uses the `DisplayCoordinates` function.

```
fXC = 1.1; fYC = -3.02;
...
DisplayCoordinates (fYC, fXC);           // incorrect usage
```

While the coordinates will be displayed, the display is not correct. There is a one-to-one correspondence between the arguments in the calling program and the parameters in the defined function.

When function arguments are used, the arguments can be passed as values, or as references, or as a pointer. In this chapter we will discuss the first two.

Calling by Value: Consider line 11 in Example 4.1.1 where the function `IsEven` is invoked as `IsEven(nNumber)`. When the control is passed to the `IsEven` function a copy of the variable `nNumber` (initialized with its current value) is created on the stack before the statements in the function are executed. The stack is a special area of memory that the compiler uses for storing named variables. The programmer is relieved of the responsibility of managing this space. Once all the statements in the function are executed and control is passed back to the calling function, the variables associated with the function are destroyed automatically. What is the implication of such a behavior? Consider the following program segments.

```
void AnalyzeData ()
{
    ...
    double dXCoor = 1.2, dYCoor = 2.4, dMaxCoor;
    dMaxCoor = MaxCoor (dXCoor, dYCoor);
    std::cout << "Max of " << dXCoor << " and " << dYCoor << " is "
              << dMaxCoor;
    ...
}
...
double MaxCoor (double d1, double d2)           // badly written function
{
```

```

    if (d2 > d1) d1=d2;
    return d1;
}

```

Will we see the output as

```

Max of 1.2 and 2.4 is 2.4
or as

```

```

Max of 2.4 and 2.4 is 2.4?

```

We will see the correct result in spite of the badly written `MaxCoor` function. This is because in C++, by default, arguments are passed as values. In this example, a copy of `d1` and a copy of `d2` are created on the stack just before the `MaxCoor` function is executed. These copies are then used in the `MaxCoor` function and the final values (whether they are changed or not) are discarded before control is passed back to the `AnalyzeData` function. In other words, the value of `dXCoor` is restored to 1.2 after control is transferred to `AnalyzeData` function so that the `std::cout` statement is executed. How can we encourage better programming practices in situations like this? A better way of defining and writing the `MaxCoor` function is to use the `const` qualifier.

```

double MaxCoor (const double d1, const double d2)
{
    if (d2 > d1)
        return (d2);
    else
        return d1;
}

```

If the function is defined with the `const` qualifier for both `d1` and `d2`, the values of these two variables cannot be changed anywhere within the function. In other words, the statement

```

if (d2 > d1) d1=d2;

```

will not compile!

Calling by Reference: A reference contains a memory address of a variable. When a function is called by reference, the memory address of the variable is passed not the value of the variable. Consider a function `swap` that should swap the values of two variables. Consider the following program segment.

```

void swap (int n1, int n2); // prototype - call-by-value
...
int nA=10, nB=20;
std::cout << "Before swap A is " << nA < " and B is " << nB << ".\n";
swap (nA, nB);
std::cout << "After swap A is " << nA < " and B is " << nB << ".\n";

```

The output before and after the call to the `swap` function is identical since the usage is call-by-value. In order to ensure that the values are swapped in the function, we need to change the program as follows.

```

void swap (int& n1, int& n2); // prototype - call-by-reference
...

```

```

int nA=10, nB=20;
std::cout << "Before swap A is " << nA < " and B is " << nB << ".\n";
swap (nA, nB);
std::cout << "After swap A is " << nA < " and B is " << nB << ".\n";

```

The ampersand character, **&**, is used to denote the memory address of a variable and hence denote that the function usage is call-by-reference. The reference symbol is used in the function declaration or prototype and in the function definition but NOT in the function call. In other words, to figure out whether the function usage is call-by-value or a call-by-reference one must look at the function prototype or function definition. The **swap** function can be written as follows.

```

void swap (int& n1, int& n2) // (int &n1, int &n2) is also correct
{
    int nTemp = n1;
    n1 = n2;
    n2 = nTemp;
}

```

Passing Vectors: So far we have seen how to pass scalar variables to a function. How do we pass a vector? C++ treats arrays differently. When an array is used (in a calling function), the contents of the entire array are passed to the function. Let's look at an example of a function that can be used to print the elements of an integer vector.

```

void PrintVector (int nV[], int nSize)
{
    for (int i=0; i < nSize; i++)
        std::cout << "Element " << i << " : " << nV[i] << "\n";
}

```

The notation **nV[]** (without a constant integer within the square parenthesis) signifies a vector. To use this function, we can define and use a vector as follows.

```

int nVBlocks[5], nVHeights[10];
...
PrintVector (nVBlocks, 5);
...
PrintVector (nVHeights, 10);

```

Note that with this example, it is possible to modify the values of the vector within the function. Since this function merely prints the values, a better function definition is

```
void PrintVector (const int nV[], int nSize)
```

that prevents modification of the elements of nV.

We will consolidate what we have learnt about functions with an example.

Example Program 4.1.2 Using vectors as function parameters

Problem Statement: Obtain several floating point values from the user. Compute and display the average value, minimum value and maximum value of these input numbers.

Solution: An examination of the problem statement shows the following components. First, we need to store several floating point values. We will store these values in a vector. Second, we need to compute three distinct values – the average, the minimum and the maximum values. We will accomplish these four tasks through the use of three functions. The function `GetValues` will be used to obtain the floating point values from the keyboard and store the values in the vector. The function `AvgValue` will be used to compute the average value of all the entries in the vector. Finally, we will use a single function `MinMaxValues` to compute the minimum and maximum values of all the entries in the vector. However, in terms of implementation, we will write two different functions `MaxValue` and `MinValue` to compute the maximum and minimum values.

main.cpp

```

1  #include <iostream>
2
3  // function prototypes
4  void ShowBanner ();
5  void GetValues (float fv[], const int nSize);
6  void AvgValue (const float fVEntries[], const int nSize,
7                  float& fAvg);
8  void MinMaxValues (const float fVEntries[], const int nSize,
9                      float& fMinV, float& fMaxV);
10 float MaxValue (const float fVV[], const int nSize);
11 float MinValue (const float fVV[], const int nSize);
12
13 int main ()
14 {
15     const int MAXVALUES = 5;
16     float fVEntries[MAXVALUES]; // stores the user input values
17
18     // print program banner
19     ShowBanner ();
20
21     // get the values from the user
22     GetValues (fVEntries, MAXVALUES);
23
24     // initialize
25     float fAvgValue;      // stores the average value
26     float f.MaxValue;    // stores the maximum value
27     float f.MinValue;    // stores the minimum value
28
29     // compute the average
30     AvgValue (fVEntries, MAXVALUES, fAvgValue);
31
32     // compute the min and max values
33     MinMaxValues (fVEntries, MAXVALUES, f.MinValue, f.MaxValue);
34
35     // display the results
36     std::cout << "\n";
37     std::cout << "DATA STATISTICS" << "\n";
38     std::cout << "-----" << "\n";
39     std::cout << "Average value : " << fAvgValue << "\n";
40     std::cout << "Minimum value : " << f.MinValue << "\n";
41     std::cout << "Maximum value : " << f.MaxValue << "\n";
42

```

```

43     return 0;
44 }
45
46 void ShowBanner ()
47 {
48     std::cout << "\tWELCOME TO EXAMPLE 4.1.2\n";
49     std::cout << "\t      S. D. Rajan      \n";
50     std::cout << "\t-----\n\n";
51 }
52
53 void GetValues (float fVV[], const int nSize)
54 {
55     int i;
56
57     // obtain the values one at a time
58     for (i=0; i < nSize; i++)
59     {
60         std::cout << "Input element [" << i+1 << "]\a : ";
61         std::cin >> fVV[i];
62     }
63 }
64
65 void AvgValue (const float fVEntries[], const int nSize,
66                 float& fAvg)
67 {
68     int i;
69
70     // initialize
71     fAvg = 0.0;
72
73     // sum all the numbers
74     for (i=0; i < nSize; i++)
75         fAvg += fVEntries[i];
76
77     // now the average
78     fAvg /= static_cast<float>(nSize);
79 }
80
81
82 void MinMaxValues (const float fVEntries[], const int nSize,
83                     float& fMinV, float& fMaxV)
84 {
85     // compute the min and max values
86     fMinV = MinValue (fVEntries, nSize);
87     fMaxV = MaxValue (fVEntries, nSize);
88
89 }
90
91 float MinValue (const float fVV[], const int nSize)
92 {
93     float fMinV;    // stores the min value
94     int i;
95
96     // set minimum value to the first entry
97     fMinV = fVV[0];
98
99     // now compare against the rest of the entries
100

```

```

101     for (i=1; i < nSize; i++)
102     {
103         if (fVV[i] < fMinV) fMinV = fVV[i];
104     }
105
106     return fMinV;
107 }
108
109 float MaxValue (const float fVV[], const int nSize)
110 {
111     float fMaxV;      // stores the max value
112     int i;
113
114     // set maximum value to the first entry
115     fMaxV = fVV[0];
116
117     // now compare against the rest of the entries
118     for (i=1; i < nSize; i++)
119     {
120         if (fVV[i] > fMaxV) fMaxV = fVV[i];
121     }
122
123     return fMaxV;
124 }
```

Let's examine the implementation of the functions. The `ShowBanner` function merely displays a banner when the program execution starts. We have used special characters in the program as shown in lines 48-50, and line 60. Their meanings along with other special characters are shown in Table 4.1.1.

Table 4.1.1 Special character sequences

Character Sequence	Remarks
\n	Newline. Positions the cursor to the beginning of the next line.
\t	Tab character. Positions the cursor to the next tab location.
\"	Double quote. To output the " character verbatim.
\\\	Backslash. To output the \ character verbatim.
\r	Carriage return. Positions the cursor to the beginning of the current line.
\a	Alert. Sounds the bell using the computer's speakers.

The `GetValues` function has two parameters. The first parameter is a floating point vector passed as reference. The second parameter is an integer that is the size of the vector. The `const` qualifier is used since it would be invalid to change the size of the vector in the function. In the `AvgValue` function, the input parameters to the function are the vector containing the values and the size of the vector, and the output from the function is the average value. The first two parameters are declared with the `const` qualifier. Hence, the values of the elements of the vector `fVEntries` and the size in `nSize` cannot be modified in the function. However, the last parameter is passed as a reference since the average value is computed in the function and the value must be set in the function. In line 30, this parameter is declared as `fAvgValue` and in the function the corresponding parameter is declared as `fAvg`. The `MinMaxValues` function has as input the vector of entries as the first parameter and the size of the vector, and has output the minimum and the maximum values. Since the minimum and maximum values are set in the function, they are passed as references. This function calls two other functions `MinValue` and `MaxValue` to obtain the actual computed min and max values.

One can also pass partial contents of a vector to a function. Consider the following example.

```
int nVBlocks[5]={1,2,3,4,5};
...
PrintVector (nVBlocks, 5); // prints all the five values
```

On the other hand, if one wishes to print only the last three values, the corresponding call would be as follows.

```
PrintVector (&nVBlocks[2], 3); // prints the last three values
```

Note the `&` before `nVBlocks`¹. There is a difference between the above call and

```
PrintVector (nVBlocks[2], 3); // will not compile!
```

If a specific element of the vector is used, e.g. `nVBlocks[2]`, then the implication is that just the third element of that vector is to be used. For example, if we wished to swap the contents of the third and the fifth elements, then the swap function can be used as follows.

```
void swap (int& n1, int& n2); // prototype - call-by-reference
...
swap (nVBlocks[2], nVBlocks[4]);
```

4.2 More About Functions

As we have seen in this chapter, functions are extremely useful especially if either the functionality of a task can be clearly identified or if functionality is required at several places in a program.

Default arguments: When one or more arguments are used in call by value, a default value can be specified for one or more of the arguments. Let us assume that we are computing the weight of a rectangular beam. The function prototype is as follows.

¹ We will see more about memory addressability in Chapter 8.

```
float BeamWeight (float fLength, float fHeight, float fWidth,
                  float fDensity=28000.0f);
```

What this prototype specifies is that if the function is called as follows

```
fWeight = BeamWeight (fLength, 0.03f, 0.01f);
```

then the function will automatically use the density value as 28000.0. The defined function is as follows.

```
float BeamWeight (float fLength, float fHeight, float fWidth,
                  float fDensity)
{
    return (fLength*fHeight*fWidth*fDensity);
}
```

Note that the function definition does not have the default value defined unlike the prototype. As an added restriction, the default values must be on the rightmost parameters. In other words

```
float BeamWeight (float fLength, float fHeight, float fWidth=0.03f,
                  float fDensity=28000.0f);
```

is valid but

```
float BeamWeight (float fLength, float fHeight=0.03f, float fWidth,
                  float fDensity=28000.0f);
```

is not. Use of default arguments must be done with care.

Overloading functions. It is possible to have two or more different definitions for functions with the same name. As we will see in the next example, we can write different functions that compute the minimum value of all the elements of a vector as we saw in Example 4.1.2 but also obtain the values of the elements of the vector via keyboard input. Here are the function prototypes for the `GetValues` function.

```
void GetValues (int nVV[], const int nSize);
void GetValues (float fVV[], const int nSize);
```

Note that while the function names are the same, the parameters are not the same. If two functions have the same name, the compiler is able to differentiate between their usage in a program based solely on the parameters (not the return value) – either the data types of the parameters must be different or the number of parameters must be different.

The primary advantage to function overloading is program readability – we do not have to concoct new names for functions even though these functions are almost always identical. However, the disadvantage is that we need to maintain several versions of the same function. We will see how to overcome this disadvantage using templates.

We illustrate these new ideas about functions in the following examples.

Example Program 4.2.1 Overloaded Functions

Problem Statement: Write a program to obtain a number of integer values and a number of double values. Store the numbers in (separate) vectors, and compute the minimum value in each vector.

Solution: We will essentially reuse the functions from Example Program 4.1.2, making appropriate changes.

main.cpp

```

1  #include <iostream>
2
3  // overloaded function prototypes
4  void GetValues (int nVV[], const int nSize);
5  void GetValues (double dVV[], const int nSize);
6  int   MinValue (const int nVV[], int nSize);
7  double MinValue (const double dVV[], int nSize);
8
9  int main ()
10 {
11     const int NUMELEMENTS = 5;
12
13     // integer values
14     int nValues[NUMELEMENTS];
15     std::cout << "Input " << NUMELEMENTS << " integers.\n";
16     GetValues (nValues, NUMELEMENTS);
17     int nMinValue = MinValue (nValues, NUMELEMENTS);
18     std::cout << "Minimum value is : " << nMinValue << "\n";
19
20     // double values
21     double dValues[NUMELEMENTS];
22     std::cout << "Input " << NUMELEMENTS << " doubles.\n";
23     GetValues (dValues, NUMELEMENTS);
24     double dMinValue = MinValue (dValues, NUMELEMENTS);
25     std::cout << "Minimum value is : " << dMinValue << "\n";
26
27     return 0;
28 }
29
30 void GetValues (int nVV[], const int nSize)
31 {
32     int i;
33
34     // obtain the values one at a time
35     for (i=0; i < nSize; i++)
36     {
37         std::cout << "Input element [" << i+1 << "]\a : ";
38         std::cin >> nVV[i];
39     }
40 }
41
42 void GetValues (double dVV[], const int nSize)
43 {
44     int i;
45
46     // obtain the values one at a time
47     for (i=0; i < nSize; i++)
48     {

```

```

49         std::cout << "Input element [" << i+1 << "]\a : ";
50         std::cin >> dVV[i];
51     }
52 }
53
54 int MinValue (const int nVV[], int nSize)
55 {
56     int nMinV;      // stores the min value
57     int i;
58
59     // set minimum value to the first entry
60     nMinV = nVV[0];
61
62     // now compare against the rest of the entries
63     for (i=1; i < nSize; i++)
64     {
65         if (nVV[i] < nMinV) nMinV = nVV[i];
66     }
67
68     return nMinV;
69 }
70
71 double MinValue (const double dVV[], int nSize)
72 {
73     double dMinV;      // stores the min value
74     int i;
75
76     // set minimum value to the first entry
77     dMinV = dVV[0];
78
79     // now compare against the rest of the entries
80     for (i=1; i < nSize; i++)
81     {
82         if (dVV[i] < dMinV) dMinV = dVV[i];
83     }
84
85     return dMinV;
86 }
```

Note that both the forms of the function `GetValues` are essentially the same except for the usage of different data types. Similarly for the `MinValue` functions. There are other issues that we need to be aware of concerning function overloading. For example, what will happen if the following statements are used in the program to invoke the `MinValue` as

```

float fValues[NUMELEMENTS], fMinV;
fMinV = minValue (fValues, NUMELEMENTS);
```

We will get a compilation error since a version of the function that supports float values does not exist.

Type Coercion in Argument Passing and Return Value from a Function

In Chapter 3 we saw data coercion issues dealing with expression evaluations. Such issues can also arise with passing arguments to a function and return values from functions. While data promotion implies

widening of the data stored in a constant or a variable, data demotion is just the opposite. Consider the following assignment.

```
nA = fX;
```

where we have a `int` on the left and a `float` on the right. In this case, the fractional part of `fX` is lost when the value is stored in `nA` provided the integral part can be stored in `nA`. Similar corruption of data can arise storing data in a `long` variable into an `int`, or a `double` into a `float`.

Now, consider the following function defined in a program.

```
double MassDensity (double dMass, double dVolume)
{
    return (dMass/dVolume);
}
```

If this function is called as follows

```
std::cout << "Density of new material : " << MassDensity(5601,2)
              << '\n';
```

then the C++ compiler will automatically promote the two arguments to 5601.0 and 2.0, and then carry out division with real numbers as 5601.0/2.0.

One has to be careful, however, if functions are overloaded. If we define another `MassDensity` function incorrectly as follows.

```
int MassDensity (int nMass, int nVolume)
{
    return (nMass/nVolume);
}
```

With this new overloaded function in the program, we will most likely obtain an incorrect value!

Let us look at another example involving exhaustive search that we have seen before.

Example Program 4.2.2 Exhaustive Search (revisited)

Problem Statement: The fluid flow through a trapezoidal channel is given by the following equation

$$P = \frac{A}{H} - \frac{H}{\tan \alpha} + \frac{2H}{\sin \alpha} \quad (4.2.1)$$

where P is the wetted perimeter, A is the cross-sectional area of the fluid, and the rest of the parameters are shown in Fig. 4.2.1.

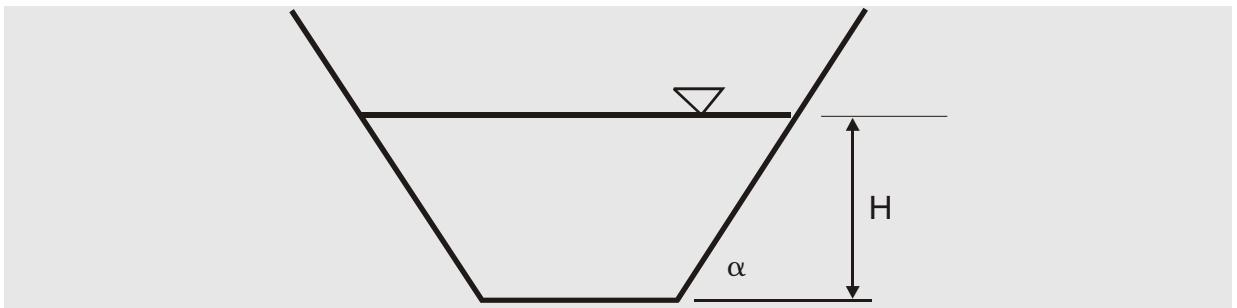


Fig. 4.2.1 Flow through a trapezoidal channel

Given the values of the wetted perimeter and the height, develop a computer program to find the values of cross-sectional area of the fluid, A and α .

Solution: The approach is not much different than what we used in Chapter 3. However, in this example, we will be using functions. We can rewrite Eqn. (4.2.1) as

$$P - \frac{A}{H} + \frac{H}{\tan \alpha} - \frac{2H}{\sin \alpha} = 0 \quad (4.2.2)$$

The basic idea is to try different values (combinations) of the unknown parameters, A and α , and find the combination that gives the value of the left-hand side of Eqn. (4.2.2) closest to zero. We can essentially call the left-hand side of Eqn. (4.2.2) as the error, δ .

Algorithm: Here is the developed algorithm.

1. Set the values (or obtain user input for) P and H . Initialize error, δ to a large value.
2. Loop through $5^\circ \leq \alpha \leq 85^\circ$ in increments of 0.01° .
3. Loop through $0.1m^2 \leq A \leq 2.0m^2$ in increments of $0.001m^2$.
4. Compute the left-hand side of Eqn. (4.2.2). If this value is smaller than δ then set δ to this value and save the values of A and α .
5. End loop for A .
6. End loop for α .
7. Print the results.

One could make the lower and upper bound for the values of A and α , and their increments as user input. In the above algorithm, the increments are deliberately chosen to be very small. The algorithm is implemented below.

main.cpp

```

1 #include <iostream>
2 #include <cmath>
3 #include <time.h>          // for time-related functions
4
5 // function prototypes
6 double UserFunction (double dP, double dA, double dH, double dAlpha);
7 void UpdateError (double dError, double& dSmallestError,
8                   double dA, double& dAAnswer,
9                   double dAlpha, double& dAlphaAnswer);
10
11 int main ()
12 {
13     time_t StartTime, EndTime;
14     double dElapsedTime;
15
16     // get the start time for the program
17     time (&StartTime);
18
19     // initialize
20     double dPerimeter = 5.0;
21     double dH = 1.0;
22     double dAlpha;           // unknown
23     double dA;              // unknown
24     double dLowerBoundAlpha = 5.0;
25     double dUpperBoundAlpha = 85.0;
26     double dAlphaIncrement = 0.01;
27     double dLowerBoundArea = 0.5;
28     double dUpperBoundArea = 2.0;
29     double dAreaIncrement = 0.001;
30     double dSmallestError = HUGE_VAL; // smallest error so far
31     double dError;           // current error
32
33     // solution
34     double dAlphaAnswer;
35     double dAAnswer;
36
37     // search for the solution
38     for (dAlpha=dLowerBoundAlpha; dAlpha <= dUpperBoundAlpha;
39          dAlpha += dAlphaIncrement)
39     {
40         for (dA=dLowerBoundArea; dA <= dUpperBoundArea;
41             dA += dAreaIncrement)
42         {
43             dError = fabs(UserFunction (dPerimeter, dA, dH, dAlpha));
44             UpdateError (dError, dSmallestError,
45                         dA, dAAnswer, dAlpha, dAlphaAnswer);
46         }
47     }
48
49     // display the solution
50     std::cout << "The obtained solution is as follows.\n";
51     std::cout << "Alpha = " << dAlphaAnswer << " degrees\n";
52     std::cout << " Area = " << dAAnswer << " m^2\n";
53
54     // get the end time for the program

```

```

56     time (&EndTime);
57
58     dElapsed = difftime (EndTime, StartTime);
59     std::cout << "    Start time in seconds since UTC 1/1/70: " <<
60         StartTime << "\n" << "    End time in seconds since UTC 1/1/70: " <<
61         EndTime << "\n" << "Elapsed time is : " << dElapsed << " s.\n";
62
63     return 0;
64 }
65
66 double UserFunction (double dP, double dA, double dH, double dAlpha)
67 {
68     const double ANGLES_TO_RADIANS = 0.0174533;
69
70     dAlpha = ANGLES_TO_RADIANS*dAlpha;
71     double dRHS = (dA/dH) - dH/tan(dAlpha) + (2.0*dH/sin(dAlpha));
72
73     return (dP - dRHS);
74 }
75
76 void UpdateError (double dError, double& dSmallestError,
77                     double dA, double& dAAnswer,
78                     double dAlpha, double& dAlphaAnswer)
79 {
80     if (dError < dSmallestError)
81     {
82         dAlphaAnswer = dAlpha;
83         dAAnswer = dA;
84         dSmallestError = dError;
85     }
86 }
```

In line 30 we initialize the value of the smallest error to a constant `HUGE_VAL` whose value is available from the math library via `cmath`. Nested `for` loops are used to vary the values of α and A in lines 38 through 48. The function `UserFunction` is called in lines 44 to compute the current error in the solution. In the next line, the function `UpdateError` is called to check for the smallest error (so far) and to save the solution – values of α and A .

The program uses the functions from C++'s time library to obtain the clock time. While not technically correct, the clock time will be indicative of the computational effort expended in the program. The data type `time_t` is a `struct` about which we will discuss in Chapter 7. The call to function `time` takes place as

```
time (&time_tVariable);
```

as this call involves call-by-pointer (discussed in Chapter 8). The `difftime` function is called in line 58 to compute the difference between two `time_t` variables.

Recursive Functions: A function that calls itself is termed a recursive function. It should be noted that any problem that can be solved using recursion can also be solved using iterations. Recursion as an option over iterations should be chosen with care. It is the preferred approach if the algorithm to solve the

problem lends itself more naturally to recursive calls. Recursive functions are computationally expensive since the overhead of function calls carries over.

As an example of recursion, let us assume that we are interested in computing the factorial of a number, n . From the definition of a factorial, we have

$$n! = (1)(2)(3)\dots(n) \quad (4.2.3)$$

with $0! = 1$. Once we rewrite the above equation as

$$n! = n(n-1)! \quad (4.2.4)$$

we can immediately see why recursion can be used to compute the factorial. The resulting code is simple.

```
long Factorial (int n)
{
    if (n == 0)      // recursion ends here
        return 1L;
    else
        return (n*Factorial (n-1));
}
```

Factorial can be programmed using iteration as follows.

```
long Factorial (int n)
{
    long lFact = 1L;
    for (int i=2; i <= n; i++)
        lFact *= i;

    return (lFact);
}
```

In this example, both the function forms are equally elegant. However, there are some situations where recursion is a clear choice.

Function calls come with execution-time overhead. The stack frame is used to keep track of the functions and the associated parameters that are used during function calls. Hence one should be careful in using functions especially if a few functions are used repeatedly in a program.

4.3 Scope of Variables

Before we discuss modular program development, it is necessary to understand what is meant by scope of a variable. Consider the following program (all statements are not included for the sake of brevity).

```
int main ()
{
    float fLength;
```

```

    cin >> fLength;
    float fArea = ComputeAreaRectangle (fLength, fLength);
    ...
    return 0;
}

float ComputeAreaRectangle (float f1, float f2)
{
    float fLength = f1;
    float fWidth = f2;
    return (fLength*fWidth);
}

```

Will the compiler issue an error message because it is unable to differentiate between the variable `fLength` declared and used in the `main` program with the variable `fLength` declared and used in the function `ComputeAreaRectangle`? The answer is “No”. Let’s review what we have learnt so far with regards to variables. A variable’s name needs to be unique within a program segment – main program or a function. In other words, two variables even if they have different data types, cannot have the same name. The scope (or life) of the variable `fLength` in the main program is from the time it is defined in the `main` program until the end of the main program signified by the `return 0` statement. Similarly, the scope of `fLength` in `ComputeAreaRectangle` is once again from its definition `float fLength = f1;` to the end of the function signified by the `return (fLength*fWidth);` statement. Such variables are called local variables. For example, `fWidth` is a local variable in the function `ComputeAreaRectangle`. Let’s look at a modified form of the function.

```

float ComputeAreaRectangle (float fLength, float fWidth)
{
    return (fLength*fWidth);
}

```

Now is the function parameter `fLength` a local variable in the `ComputeAreaRectangle` function? Yes, since it is declared in that function. Once again, the compiler will treat the two `fLength` variables (in the main program and the `ComputeAreaRectangle` function) differently since their scope limits are different.

Now consider a modified form of the previous program. Assume that the following statements appear in a single source file and compile correctly.

```

float fLength;

int main ()
{
    cin >> fLength;
    float fArea = ComputeAreaRectangle (fLength, fLength);
    ...
    return 0;
}

float ComputeAreaRectangle (float f1, float f2)
{
    float fLength = f1;

```

```

        float fWidth = f2;
        return (fLength*fWidth);
    }
}

```

Note that the variable `fLength` is declared outside of both the main program and the function `ComputeAreaRectangle`. In the main program, it appears that the variable `fLength` is used without being declared. However, according to the scope rules, a variable that is declared outside the body of the main program and all the functions, and precedes these functions, has a global scope within the file. Hence, the variable `fLength` in the main program is the global variable declared at the top of the source file. However, the variable with the same name declared in the `ComputeAreaRectangle` function is a local variable whose scope is restricted to the function. A more subtle scope rule deals with the use of variables in a compound statement. A compound statement (or block) contains one or more C++ statements within the {} braces. Consider the following statements.

```

int main ()
{
    float fA, fB;
    int n, m;
    ...
    if (fA > fB)
    {
        int i;
        i = 2*abs(n-m);
        ...
    }
    i += 2;
    ...
    return 0;
}

```

The program will not compile flagging the statement `i += 2;` as an invalid statement with the variable `i` as being undefined. The reason is that the scope of the variable `i` is entirely in the `if (fA > fB)` block. Once again it is important to remember that the scope of any variable in any block is from the time it is defined in the block until the end of the block defined by the closing brace }.

We now present an example program to illustrate the scope rules discussed so far.

Example Program 4.3.1 Understanding scope rules

Problem Statement: Obtain the perimeter and area of a segment of a circle.

Solution: The length of the arc, a of a circle is given by

$$a = r\alpha \quad (4.3.1)$$

where r is the radius of the circle and α is the arc angle in radians. Similarly, the area of a segment of a circle is given by

$$A_s = \pi r^2 \left(\frac{\alpha^\circ}{360^\circ} \right) \quad (4.3.2)$$

The program is shown below.

main.cpp

```

1  #include <iostream>
2  const float PI = 3.1415926f;           // global scope
3  const float ANGLES_TO_RADIANS = 0.0174533f; // global scope
4  float fRadius = -123.4f;               // global scope
5
6  float AreaSegmentofCircle (float fAngle, float fRadius);
7  float ArcLengthSegmentofCircle (float fAngle, float fRadius);
8
9  int main ()
10 {
11     float fRadius = 10.0f;    // local scope to the main program
12     float fArcAngle = 120.0; // local scope to the main program
13
14     std::cout << "Phase 1 of main program\n";
15     std::cout << " Value of fRadius is : " << fRadius << "\n";
16     std::cout << "Value of fArcAngle is : " << fArcAngle << "\n\n";
17
18     if (fRadius > 0.0 && fArcAngle > 0.0)
19     {
20         float fArcLength;      // scope local to the block
21         float fArea;          // scope local to the block
22         fArcLength = ArcLengthSegmentofCircle (fArcAngle, fRadius);
23         fArea = AreaSegmentofCircle (fArcAngle, fRadius);
24         std::cout << "Executing the if block\n";
25         std::cout << "Arc Length : " << fArcLength << "\n";
26         std::cout << "      Area : " << fArea << "\n\n";
27     }
28
29     std::cout << "Phase 2 of main program\n";
30     float fArcLength;      // scope local to the main program
31     float fArea;          // scope local to the main program
32     fArcLength = ArcLengthSegmentofCircle (fArcAngle, fRadius);
33     fArea = AreaSegmentofCircle (fArcAngle, fRadius);
34     std::cout << "Arc Length : " << fArcLength << "\n";
35     std::cout << "      Area : " << fArea << "\n";
36
37     return 0;
38 }
39
40 float AreaSegmentofCircle (float fAngle, float fRadius)
41 {
42     float fArea;    // local scope
43
44     fArea = PI*fRadius*fRadius*(fAngle/360.0f);
45     return (fArea);
46 }
47
48 float ArcLengthSegmentofCircle (float fAngle, float fRadius)

```

```

49  {
50      float fArcLength;    // local scope
51
52      fArcLength = (ANGLES_TO_RADIANS*fAngle)*fRadius;
53      return (fArcLength);
54 }

```

There are two declarations with initialization of the variable `fRadius` – one on line 4 and the other on line 11. The one on line 4 has a global scope meaning that the initial value of `-123.4` is available throughout the file. However, the local variable declared on line 11 overrides the global variable. Hence in the main program, `fRadius` has an initial value of `10.0` not `-123.4`. The variable `fRadius` is also declared as function parameters in the two functions. Once again, these are local variables that take precedence over the global definition. The actual value that this variable assumes when the function executes comes from the corresponding argument in the function call (lines 22, 23, 32 and 33).

Both the variables `fArcLength` and `fArea` are declared twice in the `main` program. The variables declared in lines 20 and 21, have their scopes limited to the block between lines 22 and 26. However, the scope of the variables with the same names declared on lines 30 and 31, is from line 32 through the end of the `main` program.

Let's look at a slightly revised program. What will happen if line 11 is moved after line 19? The expression on line 18 evaluates as false and the `if` block (lines 19 through 27) does not execute. Moreover, when the function calls are made in statements 32 and 33, the results are incorrect since `fRadius` has a negative value!

4.4 Developing Modular Programs

The divide and conquer philosophy is effective whether a program has 500 lines of code or 5 million lines of code. Software engineering is an evolving science and in this section we will see one of the several hundred components of software engineering and modular program development. In this section we will look at developing a program where there is more than one source file.

Preprocessing Directives: C++ provides several preprocessing directives that aid in program development and maintenance. The purpose of these directives in the source file is to tell the preprocessor (or compiler) what specific actions to perform. As per syntax rules for these directives, the number sign (#) must be the first nonwhite-space character on the line containing the directive; white-space characters can appear between the number sign and the first letter of the directive. Some directives include arguments or values. Lines containing preprocessor directives can be continued by immediately preceding the end-of-line marker with a backslash (\). Preprocessor directives can appear anywhere in a source file, but they apply only to the remainder of the source file.

We will discuss the useful ones in this section.

```
#define identifier token_string
```

The `token_string` component is optional. One or more white spaces (or blanks) must separate the `identifier` from the `token_string`. When `token_string` is defined, the `#define` directive specifies the `token_string` that is substituted for all subsequent occurrences of the `identifier` in the source file. Here are a couple of examples.

```
#define PI 3.1415926f
#define MAX(a1, a2) (a1 > a2? a1 : a2)
```

In the first case, every occurrence of `PI` in the source file is substituted with `3.1415926`. In other words, if the file contains the following statement

```
fArea = PI*fRadius*fRadius;
```

then the preprocessor turns the statement into

```
fArea = 3.1415926f*fRadius*fRadius;
```

Similarly, with the second directive, the statement

```
fUpperBound = MAX (f1, f2);
```

is converted into

```
fUpperBound = (f1 > f2? f1 : f2);
```

In C++ terminology, `MAX` is called a macro, and the preprocessor directive expands the macro. When the `#define` directive is used without a `token_string`, then all the occurrences of `identifier` are removed from the source file. The `identifier` remains defined until the source file ends or if it is undefined using the `#undef` directive. The identifier can be tested using the `#ifdef`, `#ifndef` or `#ifndef` directives.

```
#undef identifier
```

Once an identifier is defined using the `#define` directive, the identifier can be undefined using the `#undef` directive. In other words, the `#define ... #undef` directives are paired together in a zone of the source file where the identifier has a special meaning. For example,

```
#define TOLERANCE 0.0001
.....
#undef TOLERANCE
```

There is another similar directive `#defined` whose syntax is as follows.

```
#defined identifier
```

We will now examine a set of directives that are closely linked to each other.

```
#ifndef identifier
#define identifier
#if constant_expression
#else
#elif
#endif
```

The `#ifndef` (if not defined), `#ifdef` (if defined) and `#if` are all paired with the `#endif` directive. For example,

```
#ifndef HAPPY
#define HAPPY
...
#endif
```

The `constant_expression` is an integer constant expression and is made up of integer constants, character constants and the `defined` operator. For example,

```
#if DEBUGLEVEL == 0
    cout << "The value of x is " << fx << endl;
#endif
```

The `#else` and `#elif` (else if) directives go with matching `#if` and `#endif` directives. For example,

```
#if DEBUGLEVEL == 1
    cout << "The value of x is " << fx << endl;
#elif DEBUGLEVEL == 2
    cout << "The value of x is " << fx << endl;
    cout << "The value of y is " << fy << endl;
    cout << "The value of z is " << fz << endl;
#else
    cout << "Safe execution so far." << fx << endl;
#endif
```

What better way to illustrate all the ideas that we have discussed so far than through an example.

Example Program 4.4.1 A simple statistics library

Problem Statement: Develop a statistics library containing functions that support the following measures
 (a) arithmetic mean, \bar{X} , (b) median, X_m , (c) standard deviation, σ , (d) variance, σ^2 , and (e) covariance, σ_{xy} . We will assume that real numbers are used in all the computations.

Solution: Any book on statistics will provide the following formulae for the abovementioned measures.

$$\bar{X} = \frac{\sum_{i=1}^n x_i}{n} \quad (4.4.1)$$

$$X_m = \frac{x_{n/2} + x_{n/2+1}}{2} \quad \text{if } n \text{ is even} \quad (4.4.2a)$$

$$X_m = x_{(n+1)/2} \quad \text{if } n \text{ is odd} \quad (4.4.2b)$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{X})^2}{n}} \quad (4.4.3)$$

$$\sigma^2 = \frac{\sum_{i=1}^n (x_i - \bar{X})^2}{n} \quad (4.4.4)$$

$$\sigma_{xy} = \frac{\sum_{i=1}^n (x_i - \bar{X})(y_i - \bar{Y})}{n} \quad (4.4.5)$$

This statistical library (of functions) is similar to C++ math library that supports a host of functions such as `sqrt`, `fabs`, `sin` etc. These math functions can be used in any program if the function prototypes are included in the program via the `<cmath>` header file. In this example, we will create the following source files.

statpak.cpp program to use and check the functions in the statistics library

stat.h header file containing the function prototypes for the library

stat.cpp file containing the statistical functions

The advantages of splitting the entire C++ source statements into three files are many. First, we capture the entire functionality of the statistical library into two physical files – `stat.h` and `stat.cpp`. There are no extraneous issues to deal with. Second, these functions can be used in any program. One needs to include the function prototypes, typically, as

```
#include "stat.h"
```

at the top of the source file(s) in which the functions are used. In fact, we can share these functions with other programmers. Third, the process of testing and debugging becomes simpler since we have split the library functions from the actual usage of the functions. In this example, we will embed the test functionality in the source file `statpak.cpp`. Imagine that we have a large program containing

several thousand lines of source code distributed into several physical files. If a few corrections are made in a single file, is it necessary to compile all the source files? We need to recompile only those files where the source statements are changed with a modular program development. Finally, the set of actual values used to test the functions will be hardcoded into the test program. This is not quite efficient since this will involve editing and inserting new values, recompiling and linking every time we wish to try a new set of test values. We will overcome this drawback once we learn how to deal with external files (Chapter 12).

The source files are presented and discussed below.

stat.h

```

1  #ifndef __RAJAN_STAT_H__
2  #define __RAJAN_STAT_H__
3
4  // function prototypes
5  float StatMean (const float fv[], int nSize);
6  float StatMedian (const float fv[], int nSize);
7  float StatStandardDeviation (const float fv[], int nSize);
8  float StatVariance (const float fv[], int nSize);
9  float StatCoVariance (const float fvx[], const float fv[],
10                           int nSize);
11
12 #endif

```

Potential conflicts may be created when several files are used in a program and each file may include the same header file. To avoid including a header file more than once when a file is compiled, C++ provides the following mechanism. Define a unique identifier to be associated with the header file. The compiler uses this identifier to track how many times a specific header file is referenced, and loads the contents of the header file only once. In line 1, the preprocessor directive `#ifndef` is used. We have already seen and used the `#include` preprocessor directive. The `#ifndef` signifies if not defined. In other words, if the identifier following `#ifndef` has not been defined and used so far, then load and parse the statements that follow until the `#endif` directive (line 12) is encountered. In line 2, the identifier associated with the `stat.h` header file is defined using the `#define` preprocessor directive. It is a good idea to make the identifier unique so as to avoid conflicts with other identifiers defined in other source files. In this book the following naming convention is used – the first two characters are the underscore character “`_`”. Similarly, the last three characters are `H__`. The function prototypes are defined in lines 5 through 10.

stat.cpp

```

1  #include <cmath>
2  #include "stat.h"
3
4  /////////////////////////////////
5  // functions used by the stat library
6  /////////////////////////////////
7  bool IsEven (int n)
8  {
9      if ((n % 2) == 0)
10          return true;      // even number

```

```

11         else
12             return false;      // odd number
13     }
14
15     /////////////////////////////////
16     // stat library functions
17     /////////////////////////////////
18     float StatMean (const float fV[], int nSize)
19     {
20         int i;
21         float fSum = 0.0f;
22         float fMean = 0.0f;
23
24         if (nSize <= 0)
25             return (fMean);
26
27         for (i=0; i < nSize; i++)
28             fSum += fV[i];
29
30         fMean = fSum/static_cast<float>(nSize);
31
32         return (fMean);
33     }
34
35     float StatMedian (const float fV[], int nSize)
36     {
37         float fMedian = 0.0f;
38         if (nSize <= 0)
39             return (fMedian);
40
41         if (IsEven(nSize))
42         {
43             int n = nSize/2;
44             fMedian = 0.5f*(fV[n-1] + fV[n]);
45         }
46         else
47         {
48             int n = (nSize+1)/2;
49             fMedian = fV[n-1];
50         }
51
52         return (fMedian);
53     }
54
55     float StatStandardDeviation (const float fV[], int nSize)
56     {
57         int i;
58         float fStandardDeviation = 0.0f;
59
60         float fMean = StatMean (fV, nSize);
61         if (fMean == 0.0f)
62             return (fMean);
63
64         for (i=0; i < nSize; i++)
65         {
66             fStandardDeviation += static_cast<float>(pow(fV[i]-fMean, 2.0));
67         }

```

```

68
69     float fTemp = fStandardDeviation/static_cast<float>(nSize);
70     fStandardDeviation = static_cast<float>(sqrt(fTemp));
71
72     return (fStandardDeviation);
73
74 }
75
76 float StatVariance (const float fV[], int nSize)
77 {
78     float fVariance = StatStandardDeviation (fV, nSize);
79     fVariance *= fVariance;
80
81     return (fVariance);
82 }
83
84 float StatCoVariance (const float fVX[], const float fVY[],
85                         int nSize)
86 {
87     int i;
88     float fCoVariance = 0.0f;
89
90     float fMeanX = StatMean (fVX, nSize);
91     float fMeanY = StatMean (fVY, nSize);
92
93     if (fMeanX == 0.0f || fMeanY == 0.0f)
94         return (fCoVariance);
95
96     for (i=0; i < nSize; i++)
97     {
98         fCoVariance += (fVX[i]-fMeanX)*(fVY[i]-fMeanY);
99     }
100    fCoVariance /= static_cast<float>(nSize);
101
102    return (fCoVariance);
103 }

```

The functions in the library are defined in this file. Note that in line 2, the `#include "stat.h"` directive is used so that the function prototypes are available to the compiler. Compare this to the manner in which we have seen include directives before. The difference between having the file name in the angle brackets `<..>` and between the double quotes `".."` is that when the angle brackets are used, the compiler looks for the file in a special directory, whereas when the double quotes are used, the compiler looks for the file in the current directory. The rest of the file is a strict implementation of Equs. (4.4.1) through (4.4.5). The functions are reused as much as possible so as to avoid replicating the code – the standard deviation function (`StatStandardDeviation`) calls the function that computes the mean (`StatMean`). Some error checks are carried out. For example, every function checks to see whether the size of the vector is positive (non-zero). We also assume that the vectors are already sorted for the `StatMedian` function to work correctly.

statpak.cpp

```

1 #include <iostream>
2 #include "stat.h"
3

```

```

4  void ComputeStats (const float fV[], int nSize);
5
6  int main ()
7  {
8      const int SIZEA = 3;
9      const int SIZEB = 3;
10     const int SIZEC = 4;
11
12     // define and populate the vectors
13     float fVA[SIZEA] = {-5.1f, 12.3f, 33.9f};
14     float fVB[SIZEB] = {5.1f, 19.0f, 22.3f};
15     float fVC[SIZEC] = {-900.3f, -18.7f, 33.4f, 123.5f};
16
17     // compute stats for vector A
18     std::cout << "    Stats for vector A\n";
19     std::cout << "    -----'\n";
20     ComputeStats (fVA, SIZEA);
21
22     // compute stats for vector B
23     std::cout << "    Stats for vector B\n";
24     std::cout << "    -----'\n";
25     ComputeStats (fVB, SIZEB);
26
27     // compute stats for vector C
28     std::cout << "    Stats for vector C\n";
29     std::cout << "    -----'\n";
30     ComputeStats (fVC, SIZEC);
31
32     // covariance
33     float fCoVariance = StatCoVariance (fVA, fVB, SIZEA);
34     std::cout << "Covariance - vectors A and B: " <<
35             fCoVariance << "\n\n";
36
37     return 0;
38 }
39
40 void ComputeStats (const float fV[], int nSize)
41 {
42     float fMean = StatMean (fV, nSize);
43     float fMedian = StatMedian (fV, nSize);
44     float fStandardDeviation = StatStandardDeviation (fV, nSize);
45     float fVariance = StatVariance (fV, nSize);
46
47     std::cout << "          Mean: " << fMean << "\n";
48     std::cout << "          Median: " << fMedian << "\n";
49     std::cout << "          Standard Deviation: " << fStandardDeviation << "\n";
50     std::cout << "          Variance: " << fVariance << "\n\n";
51 }
```

This is the test program that is used to test and debug the statistics library. The prototypes of the function are included in line 2. In line 4 we see the prototype of a local function. Three vectors are declared and defined in lines 13 through 15. The statistical measures are computed for each vector via a call to `ComputeStats` function. Finally, in line 33 the covariance function is tested using vectors A and B.

Finally, we will look at one more form of scope that cuts across different files using the **extern** keyword. Consider the following problem. A program is being developed and the main program and all the other functions are contained in **two** source files. Suppose we have a variable **nPoints** in the first file (called **main.cpp**), and that we wish to use this variable (access and/or modify the value) in the second file (called **draw.cpp**). The schematic diagram as to how to achieve this is shown in Fig. 4.4.1.

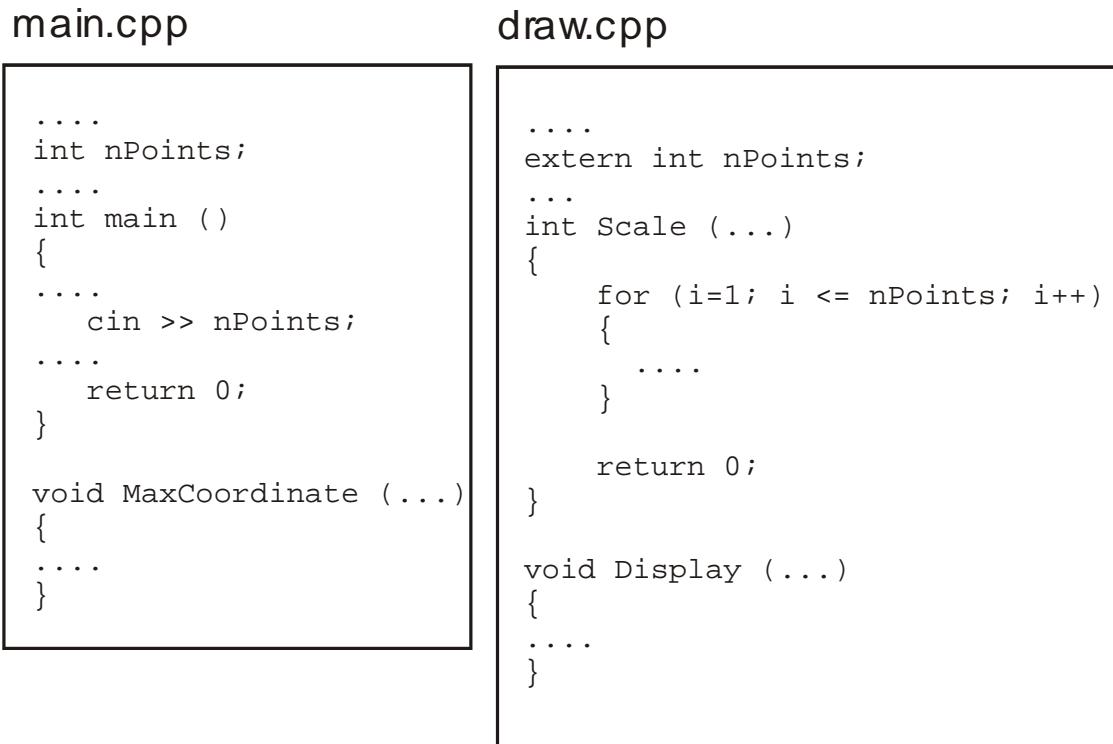


Fig. 4.4.1 An example showing how the **extern** keyword is used

A variable can have a global scope if it is declared outside all the blocks and functions (and the qualifier **static** does not appear before the variable). This must be done once and only once in all the files that are a part of the program. In the above example, this is done in the file **main.cpp**. If the variable needs to be used in another source file, then the **extern** qualifier must appear before the variable. In the above example, this is done in file **draw.cpp** and hence **nPoints** is visible in all the functions that appear in that file.

Example Program 4.4.2 Global variables via extern keyword

Problem Statement: Develop a library of vector functions – dot product and length of a vector. Keep track of the total number of floating point operations in the program.

Solution: The dot product, d between two vectors $\mathbf{a}_{n \times 1}$ and $\mathbf{b}_{n \times 1}$ is defined as follows

$$d = \sum_{i=1}^n a_i b_i \quad (4.4.6)$$

The number of floating point operations is n . The length, l of a vector $\mathbf{a}_{n \times 1}$ is defined

$$l = \sqrt{a_1^2 + a_2^2 + \dots + a_n^2} \quad (4.4.7)$$

The number of multiplications is n ; we will assume that the square root is equal to one multiplication (strictly speaking the square root is evaluated using an iterative technique where each iteration involves several multiplications).

We will develop the program in three source files. The first file (`main.cpp`) will contain the main program. In the second file (`VectorOps.h`), we will declare the two function prototypes – `VectorDotProduct` and `VectorLength`. In the third file (`VectorOps.cpp`), we will define the two functions. In order to track the number of floating point operations (FLOPS), we will use a global variable `nFLOPS` that we will initialize to zero in the main program. In the two functions, we will update the value of `nFLOPS`.

main.cpp

```

1 // include files
2 #include <iostream>
3 #include "VectorOps.h"
4
5 // global variables
6 int nFLOPs; // nFLOPs is a global variable
7
8 int main ()
9 {
10    // define vectors used in the program
11    float fX[3]={1.0f, 2.0f, 3.0f};
12    float fY[3]={4.0f, 5.0f, 6.0f};
13
14    // initialize
15    nFLOPs = 0;
16
17    // carry out the vector operations
18    int i;
19    for (i=1; i <= 10000; i++)
20    {
21        float fDP = VectorDotProduct (fX, fY, 3);
22        float fLengthX = VectorLength (fX, 3);
23        float fLengthY = VectorLength (fY, 3);
24    }
25
26    // display the # of floating point operations
27    std::cout << "Number of floating-point operations: "
28    << nFLOPs << "\n";
29
30    return 0;
31 }
```

Note the declaration of the global variable in line 6. The variable is initialized in line 15. To use the vector functions, we will define and declare the vectors on lines 11 and 12, and use them in the `for` loop (lines 19 through 24).

VectorOps.h

```

1  #ifndef __RAJAN__VECTOROPS_H_
2  #define __RAJAN__VECTOROPS_H_
3
4  float VectorDotProduct (const float fA[], const float fB[],
5                           int nSize);
6  float VectorLength (const float fA[], int nSize);
7
8  #endif

```

To ensure that the header file containing the prototypes is included during the program compilation only once, the `#ifndef .. #define .. #endif` statements are used.

VectorOps.cpp

```

1 // include files
2 #include <cmath>
3
4 // global variables
5 extern int nFLOPs; // nFLOPs is defined somewhere else
6 // in the program
7
8 float VectorDotProduct (const float fVA[], const float fVB[],
9                         int nSize)
10 {
11     int i;
12     float fSum=0.0f;
13
14     for (i=0; i < nSize; i++)
15         fSum += fVA[i] * fVB[i];
16
17     nFLOPs += nSize;
18
19     return (fSum);
20 }
21
22 float VectorLength (const float fV[], int nSize)
23 {
24     int i;
25     float fLength = 0.0f;
26
27     for (i=0; i < nSize; i++)
28     {
29         fLength += fV[i] * fV[i];
30     }
31     fLength = static_cast<float>(sqrt(fLength));
32
33     // take square root as one FLOP
34     nFLOPs += nSize+1;

```

```

35
36     return (fLength);
37 }

```

The global variable nFLOPS is declared in line 5 but note that the `extern` qualifier is used. The values are updated in line 17 in function `VectorDotProduct` and in line 34 in function `VectorLength`. As a note of caution, we will minimize (if not eliminate) the use of global variables (see Programming Tip #12).

Storage Classes

C++ has several storage classes (not to be confused with scope). When a variable is declared and used, memory needs to be allocated and subsequently deallocated when the variable goes out of scope. In other words, the storage class determines the memory life of a variable and a function. We will discuss some of the storage classes here.

automatic variables

A local variable's life is defined by its scope – the variable is created when and where the variable is declared in a block and ends when execution exits the block. However, C++ automatically assumes the responsibility of allocating and deallocating the memory for the variable. Hence the name for the storage class – automatic. There is a C++ keyword `auto` that can be used as follows

```
auto int nPopulation;
```

However, in this book we will not use the keyword.

static variables

Global and static variables and functions exist from the time a program starts execution till the time the program finishes execution. We saw the use of the `extern` keyword before in connection with global variables. The keyword can also be used with functions. Similarly, the keyword `static` can be used with variables and functions. Here is an example of a static variable used in a function.

```

void DoNothing ()
{
    static int nCount = 0;
    nCount++;
    std::cout << "Count is : " << nCount << "\n";
}

```

We will use this function from another function as follows.

```
for (int i=0; i < 4; i++)
    DoNothing ();
```

When the program is executed, the following output is generated.

Fig. 4.4.2 Output from a function using a `static` variable

Tip: Finally, let's review the type of scope rules as they apply to identifiers – local scope is within a block contained between the curly braces { . . . }, function scope is only within a function not outside the function, file scope is applicable if the identifier appear outside all the blocks and functions (if the qualifier `static` does not appear before the identifier then the identifier is global), and prototype scope begins and ends with a prototype statement.

4.5 Function Templates

We were introduced to the concept of function overloading in Section 4.2. We immediately saw the advantages to naming functions based on their functionalities rather than the data type associated with the function parameters. As we saw in Example Program 4.2.1, the two functions to compute the minimum value of a set of numbers were exactly the same except for the data type associated with the set of numbers – one was `int` and the other `double`. Clearly if we wanted to support the other primitive data types, then we would need a version for `short`, `long`, and `float` data types – a total of 5 different functions. This can lead to severe program maintenance problems. For example, even a small change in the algorithm to compute the minimum value would involve making the same change to all the 5 functions. C++ provides a cleaner approach using a function template.

A function template is a complete function where the basic functionality is coded without explicitly specifying the data type of some or all of the function parameters, so as to create a family of functions. We will now discuss the template syntax. The first line for a template function is as follows.

```
template <class T>
```

The line is called the template prefix. The keywords `template` and `class` appear as shown above. The parameter `T` is the type parameter. The compiler will substitute the appropriate data type for `T` based on the function call. The parameter name can be any legal identifier. We have chosen `T` for convenience sake. Also, one could have more than one parameter separated by commas. The general syntax is

```
template < [typeList] [, [argList]] > declaration
```

and we will see this declaration and usage later when template classes are introduced in Chapter 9. Let's go back to Example Program 4.2.1 and the `int` version of the function `MinValue`.

```

int MinValue (const int nVV[], int nSize)
{
    int nMinV;      // stores the min value
    int i;

    // set minimum value to the first entry
    nMinV = nVV[0];

    // now compare against the rest of the entries
    for (i=1; i < nSize; i++)
    {
        if (nVV[i] < nMinV) nMinV = nVV[i];
    }

    return nMinV;
}

```

We could easily rewrite this function substituting the `int` data type associated with the vector of number with a generic data type `T` as follows.

```

template <class T>
T MinValue (const T TVV[], int nSize)
{
    T TMinV;      // stores the min value
    int i;

    // set minimum value to the first entry
    TMinV = TVV[0];

    // now compare against the rest of the entries
    for (i=1; i < nSize; i++)
    {
        if (TVV[i] < TMinV) TMinV = TVV[i];
    }

    return TMinV;
}

```

Compare the differences between the two versions. In the template version, we have the special first line. The function definition for

```
int MinValue (const int nVV[], int nSize)
```

now reads

```
T MinValue (const T TVV[], int nSize)
```

Beyond this, the appropriate `int` declarations are substituted with `T`! All the other changes are style changes (writing `TVV` instead of `nVV` etc.).

Example Program 4.5.1 Function templates (Example Program 4.2.1 revisited)

Problem Statement: Write a program to obtain a number of integer values and a number of double values. Store the numbers in (separate) vectors, and compute the minimum value in each vector.

Solution: We will make the original program much shorter and cleaner using function templates. For both the functions – GetValues and MinValue. In addition, we will split the source files by putting the functions in a separate file. The main program is shown below.

main.cpp

```

1 #include <iostream>
2 #include "templates.h"
3
4 int main ()
5 {
6     const int NUMELEMENTS = 5;
7
8     // integer values
9     int nValues[NUMELEMENTS];
10    std::cout << "Input " << NUMELEMENTS << " integers.\n";
11    GetValues (nValues, NUMELEMENTS);
12    int nMinValue = minValue (nValues, NUMELEMENTS);
13    std::cout << "Minimum value is : " << nMinValue << "\n";
14
15     // double values
16     double dValues[NUMELEMENTS];
17     std::cout << "Input " << NUMELEMENTS << " doubles.\n";
18     GetValues (dValues, NUMELEMENTS);
19     double dMinValue = minValue (dValues, NUMELEMENTS);
20     std::cout << "Minimum value is : " << dMinValue << "\n";
21
22     return 0;
23 }
```

The source statements for the main program are identical to Example Program 4.2.1 except for line 2 where the `#include` statement is used to declare the two functions used in the program.

Tip: If the template functions are defined in a separate file then the source statements need to be included in the header file (not a C++ source file) as shown below.

templates.h

```

1 #ifndef __RAJAN__TEMPLATES_H__
2 #define __RAJAN__TEMPLATES_H__
3
4 template <class T>
5 void GetValues (T TVV[], const int nSize)
6 {
7     int i;
8
9     // obtain the values one at a time
10    for (i=0; i < nSize; i++)
11    {
12        std::cout << "Input element [" << i+1 << "]\a : ";
13        std::cin >> TVV[i];
14    }
15 }
16
```

```

17  template <class T>
18  T MinValue (const T TVV[], int nSize)
19  {
20      T TMinV;      // stores the min value
21      int i;
22
23      // set minimum value to the first entry
24      TMinV = TVV[0];
25
26      // now compare against the rest of the entries
27      for (i=1; i < nSize; i++)
28      {
29          if (TVV[i] < TMinV) TMinV = TVV[i];
30      }
31
32      return TMinV;
33  }
34
35 #endif

```

The template functions are almost identical to the non-template functions except for the differences pointed out earlier (and they are defined in a header file). The entire program is now more compact.

4.6 Case Study: A 4-Function Calculator

In this section we will develop a simple four-function calculator. The user will be prompted as

```
Enter +-*/CS or value [???
```

where ??? is the current value that one would see on the calculator. The four functions are add, subtract, multiply and divide as signified by the following symbols **+**, **-**, *****, **/**. To clear the display (reset the value to zero), we will use **C** or **c**. Similar to power off the calculator, we will use the symbol **S** or **s**. Let us look at some examples of using the calculator. To compute $12(5.1 + 6.3)$, the user of the program would do the following.

```

Enter +-*/CS or value [0] 5.1
Enter +-*/CS or value [5.1] +
Enter +-*/CS or value [5.1] 6.3
Enter +-*/CS or value [11.4] *
Enter +-*/CS or value [11.4] 12
Enter +-*/CS or value [136.8] S

```

We will try to resolve an important problem before starting to develop the algorithm and program structure. One of the biggest problems that new C++ programmers have with the standard input class is the manner in which input is read and interpreted. For example, what happens with the following code

```
#include <iostream>
```

```

int main ()
{
    int nPoints;

    std::cout << "Input number of points: ";
    std::cin >> nPoints;
    std::cout << "You have input: " << nPoints << "\n";

    return 0;
}

```

if the user types an invalid value such as 1w3 instead of 123. This is the output generated by the Microsoft Visual C++ compiler is shown in Fig. 4.6.1.

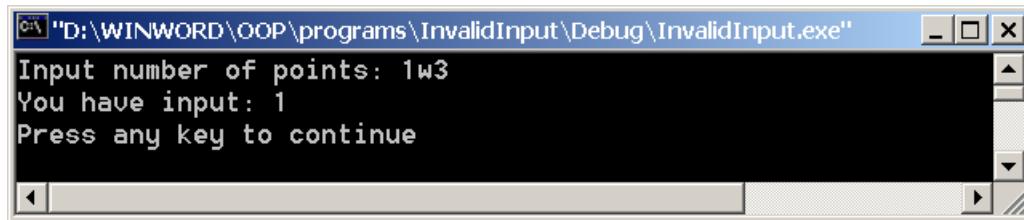


Fig. 4.6.1 Output generated by an invalid input

The standard input class parses the input buffer and stops when it encounters an invalid character. What is worse is that the rest of the input **w3** still remain in the input buffer and will be read in by any subsequent **cin** statements. There are exhaustive mechanisms available within the standard input class to check whether the **read** statement executed successfully. We have created one such function that is described below.

GetInteractive Function: The prototypes for the **GetInteractive** functions are shown below.

```

// int version
void GetInteractive (const string& szString, int& V);
void GetInteractive (const string& szString, int& V,
                    int nL, int nU);

// long version
void GetInteractive (const string& szString, long& V);
void GetInteractive (const string& szString, long& V,
                     long lL, long lU);

// float version
void GetInteractive (const string& szString, float& V);
void GetInteractive (const string& szString, float& V,
                     float fL, float fU);

// double version
void GetInteractive (const string& szString, double& V);
void GetInteractive (const string& szString, double& V,
                     double dL, double dU);

// string version
void GetInteractive (const string& szString, string& V,
                     int n);

```

```
// raw access and conversion functions
int GetLongValue (string& szInput, long& lv);
int GetDoubleValue (string& szInput, double& dv);
```

For example we could rewrite the previous program as follows. The main program is shown below and the entire program would be formed by an additional source code contained in `getinteractive.cpp`.

```
#include "getinteractive.h"

int main ()
{
    int nPoints;

    GetInteractive ("Input number of points: ", nPoints);
    std::cout << "You have input: " << nPoints << "\n";

    return 0;
}
```

The overloaded versions can be used if range checking is to be performed. For example, what if the value of `nPoints` is between 1 and 100. Then we would rewrite the call to `GetInteractive` as.

```
GetInteractive ("Input number of points: ", nPoints, 1, 100);
```

The details of the `GetInteractive` function are not discussed here. Interested readers can explore the source code (`getInteractive.cpp`).

And now on to the development of the algorithm. We will store the displayed value in a variable called `dMemory` and the next input value in a variable called `dNext`. We will track the operation to be performed using a variable called `nOper` that will have a value of 1, 2, 3 and 4 if binary addition, subtraction, multiplication and division is to be performed; otherwise the value will be set to 0. To obtain and act on the user command, we will use a variable called `nCommand` that will have the following values – 0 to exit the program, 1 for addition, 2 for subtraction, 3 for multiplication, 4 for division, 5 to clear the display to zero, and 6 if the user has input a number.

Algorithm

1. Initialize `dMemory`, `dNext` and `nOper` to zero.
2. Loop until user terminates the program.
3. Get user command. Limit number of input characters to 10.
4. Is input one of `+-*/cCsS` or is it a number? Try to read the number as a double precision value. If there is an invalid input, ask the user for a valid input.
5. If the input is to stop the calculations, then exit the program.

6. If the input is to clear the memory, then set `dMemory` and `nOper` to zero.
7. If the input is one of the four operators, update `nOper`.
8. Else the input is a number. Check `nOper`. If `nOper` has been defined (`nOper` is not zero) then carry out the operation, update `dMemory` and set `nOper` is zero. Else store this number in `dNext`.
9. End loop.
10. Terminate program.

The program is developed in a modular form with all the steps except Steps 3 and 4 implemented in the main program (main.cpp). Steps 3 and 4 are implemented in several functions (utility.h and utility.cpp) and the details are presented below.

Example Program 4.6.1 A Four-Function Calculator Source Code

main.cpp

```

1   #include "utility.h"           // contains function prototypes
2
3   int main()
4   {
5       // show program banner
6       ShowBanner ();
7
8       // initialize
9       double dMemory = 0.0;    // contains current displayed value
10      double dNext = 0.0;     // new user input value
11      int nOper = 0;         // operation to be performed
12
13      // loop until user exits the program
14      for (;;)
15      {
16          // get user command
17          int nCommand = UserCommand (dNext, dMemory);
18
19          // stop the program?
20          if (nCommand == 0)
21              break;
22
23          // branch on user command
24          switch (nCommand)
25          {
26              case 1:    // addition
27                  nOper = 1;
28                  break;
29              case 2: // subtraction
30                  nOper = 2;
31                  break;
32              case 3: // multiplication
33                  nOper = 3;

```

```

34         break;
35         case 4: // division
36         nOper = 4;
37         break;
38         case 5: // clear the memory
39         dMemory = 0.0;
40         nOper = 0;
41         break;
42         case 6: // a number
43             // operation defined in previous input?
44             // yes.
45             if (nOper == 1)
46                 dMemory = Add (dMemory, dNext);
47             else if (nOper == 2)
48                 dMemory = Subtract (dMemory, dNext);
49             else if (nOper == 3)
50                 dMemory = Multiply (dMemory, dNext);
51             else if (nOper == 4)
52                 dMemory = Divide (dMemory, dNext);
53             else // no. store the number
54                 dMemory = dNext;
55
56             // invalidate current operation in anticipation
57             // of a new operation
58             if (nOper >= 1 && nOper <= 4)
59                 nOper = 0;
60         }
61     }
62
63     // sign off
64     ShowGoodbye ();
65
66     return 0;
67 }
```

There are seven utility functions. The ShowBanner and ShowGoodBye functions are used once at the beginning and at the end of the program. The UserCommand function parses the user input and returns only if the input is valid. The binary operations are carried out in functions Add, Subtract, Multiply and Divide. The function prototype file is shown below.

utility.h

```

1  #ifndef __RAJAN.Utility_H__
2  #define __RAJAN.Utility_H__
3
4  void ShowBanner ();
5  void ShowGoodbye ();
6  int UserCommand (double&, const double);
7
8  double Add      (const double, const double);
9  double Subtract (const double, const double);
10 double Multiply (const double, const double);
11 double Divide   (const double, const double);
12
13 #endif
```

The implementation of the utility functions is shown below.

utility.cpp

```

53         if (GetDoubleValue (szCommand, dv) == 0) // valid number?
54     {
55         dNext = dv;
56         nCommand = 6;
57     }
58 }
59
60 // illegal input
61 if (nCommand == -1)
62     std::cout << "\nInvalid input.";
63 else
64     bError = false;
65
66 } while (bError == true); // loop until input is legal
67
68 return nCommand;
69 }
70
71 // these functions implement binary + - * /
72 double Add (const double dN1, const double dN2)
73 {
74     return (dN1 + dN2);
75 }
76
77 double Subtract (const double dN1, const double dN2)
78 {
79     return (dN1 - dN2);
80 }
81
82 double Multiply (const double dN1, const double dN2)
83 {
84     return (dN1 * dN2);
85 }
86
87 double Divide (const double dN1, const double dN2)
88 {
89     // check for divide by zero
90     double dResult = (dN2 != 0.0? dN1 / dN2 : 0.0);
91     return (dResult);
92 }
```

The most interesting part of the implementation deals with formatting the user prompt. Recall that the user prompt must be of the form

```
Enter +-*/CS or value [???
```

The first part of the string is straightforward. The only sticky part is how to get the current value of `dMemory` instead of `???` and store the entire prompt as a standard string. We will study more about strings in Chapter 7. However, we will explain what is necessary to format strings. First, we need to include the class to format strings. Recall that we have been using the `iostream` for input and output. Similarly, the `sstream` class links strings with streams. In other words, it is possible to read from or write to a string using the formatting capabilities that we have seen in the stream classes. In lines 2 and 3, we declare the string stream class and indicate that we wish to use the standard output string stream class in the program. The actual variable `szPrompt` associated with this class is declared

on line 28. Note how the variable (or object) is used to format the string on lines 30 and 31. If we had wanted to write the string to the standard output, we would have used the following statement

```
std::cout << "\nEnter +-*/CS or value [ " << dMemory
<< " ] ";
```

The first parameter of the GetInteractive function is declared as `const string&`. The conversion from `ostringstream` into a `const string&` requires the use of `str` member function from the `ostringstream` class as we see in line 34. As we have mentioned before, sometimes it is easier to use and then understand why. We have used the advanced features here only because there is no other alternative. The “why” will be tackled in Chapter 7.

Second, we have used an effective but not elegant approach to taking care of the divide by zero problem in the Divide function. In line 90, if the denominator is zero, we merely return a zero value as the result instead of flagging the error. What is an alternative? We could issue an error message and terminate the program. We will see more about error handling in the next section and throughout the rest of the book.

4.7 Exception Handling

C++’s utilities to handle errors or exceptional situations is called exception handling. Examples of exceptional situations include result of a math computation that is outside the range of numbers that can be represented (underflow or overflow), dividing by zero, invalid math operation such as taking the square root of a negative number, accessing an invalid element of an array, reading from a file that does not exist etc.

try/catch Block

Exception handling is made up of the `try/catch` block. The `try` block contains C++ statements that may potentially generate an exception as well as statements that should be executed if no exception occurs. The process of generating an exception is called throwing an exception. One or more `catch` blocks immediately follow the `try` block. The general format is as follows.

```
try
{
    // statements including at least one of the following
    throw expression;
    ...
}
catch (datatype_1 identifier)
{
    // exception handling statements
}
...
catch (datatype_n identifier)
{
    // exception handling statements
}
catch (...)
```

```

{
    // exception handling statements
}

```

Each `catch` block specifies a unique datatype that it can catch and contains an exception handler. The last `catch` block contains three dots (and no data type) and can be used to catch any type of exception not caught by the preceding `catch` block(s). The identifier associated with the `catch` block is the catch block parameter and is designed to catch the exception thrown by the `try` block.

Example Program 4.7.1 Exception Handling

The following program is a simple one. The user inputs a floating point number and the program computes its reciprocal and its square root. In this example we will see how to catch three types of errors and handle them. The first is catching an invalid input – user typing a number that is not a floating point number. The second error is if the user inputs a zero value since it is not possible to compute its reciprocal. The last error is if the user inputs a negative number since we are interested in computing a real square root.

```

1 #include <iostream>
2 #include <string>
3 #include <cmath>
4
5 int main ()
6 {
7     double dUser;    // user input value
8     std::string szReadError ("Invalid user input.\n");
9
10    // try-catch block to read and interpret user input
11    try
12    {
13        std::cout << "Input a real positive number: ";
14        std::cin >> dUser;
15
16        if (std::cin.fail())
17            throw szReadError;
18        else if (dUser <= 0.0)
19            throw dUser;
20
21        // no errors encountered
22        std::cout << "Reciprocal of " << dUser << " is " << 1.0/dUser   << '\n';
23        std::cout << "Square root of " << dUser << " is " << sqrt(dUser) << '\n';
24    }
25    catch (double dx)
26    {
27        if (dx == 0.0)
28            std::cout << "Cannot divide by zero.\n";
29        else
30            std::cout << "Cannot find square root of a negative number.\n";
31    }
32    catch (const std::string& szMessage)
33    {
34        std::cout << szMessage;
35    }
36    catch (...) // catch all block
37    {
38        std::cout << "Unknown error.\n";
39    }
40
41    return 0;
42 }

```

The `try` keyword is in line 11 and the `try` block is contained in lines 12 through 24. The user input is read in line 14. The validity of the input is checked in line 16 through the `fail()` member function. If the statement is true (user entered an invalid input), an exception with a `std::string` data type is thrown in line 17. The other two errors are detected in line 18 and the associated exception is thrown in line 19 with a `double` data type. If no error is detected, the reciprocal and square root values are computed and displayed on the console in lines 22 and 23.

The first `catch` block with a `double` data type is contained in lines 25 through 31. Appropriate statements are output to the console indicating the type of error (zero value or a negative value). The second catch block with a `std::string` data type is contained in lines 32 through 35. The exception handler outputs an error message that the user input is invalid. Finally, the catch-all block is contained in lines 36 through 39 and should never be executed. It is merely shown in this program to illustrate how a catch-all block may be used in a program.

We will look at C++ provided exception classes in Chapter 9.

Summary

Material from the first four chapters should enable a programmer to completely write moderately complex programs. However, most programmers do not write complete programs. Object-oriented programming makes it possible to use and modify components written by others. We will start the study of numerical analysis from the next chapter.

While there is no universal programming style, we will set the guidelines for some good programming practices.

Programming Style Tip 4.1: Naming variables, constants and functions

Variable names in C++ start with an alphabet and involve the following characters: 0-9, a-z, A-Z, \$ and _. In this book, we will maintain the following convention. The objective is to make the naming of the variables uniform and predictable. C++ provides the following basic scalar types – integer, float, double, boolean, and character. In addition, C++ also has support for vectors and matrices as a basic type and through STL (standard template library). However, we will use our own vector and matrix templates, as we will see in Chapter 9. Whenever appropriate, we will use the **string** class to store character strings.

Data Type	Variable Name Prefix	Examples
Integer scalar	n?	nX, nIterations, nJoints
Float scalar	f?	fY, fTolerance
Double scalar	d?	dArea, dP123
Boolean scalar	b?	bDone, bConverged
Integer vector	nV?	nVScores, nVSSN
Integer matrix	nM?	nMVertices, nMShapes
Float vector	fV?	fVRHS, fVForces
Float matrix	fM?	fMCoeff, fMElementForces
Double vector	dV?	dVRHS, dVGPA
Double matrix	dM?	dMCoeff, dMCoordinates
String	sz?	szNames, szStates

Class name	C?	CNode, CPoint
Pointer	p?	pVCoor
Member of class	m_?	m_nRows, m_fVCoordinates

We will denote constants with upper case alphabets. For example

```
const float PI=3.1425926f;
```

We will denote function names starting with a capitalized alphabet, e.g. void ReadInput (...).

Tread lightly with this tip. Extreme use (not overuse) of this tip may lead to undecipherable variable names such as pVdMCoordinates.

Programming Style Tip 4.2: Comments

A good programming habit is to liberally sprinkle the source code for a computer program with comments that are meaningful. One approach is to make some of the comments mirror the steps from a detailed algorithm or from an associated theory. We have seen examples of this approach in this chapter.

Programming Style Tip 4.3: Use of blank spaces and blank lines

Use blank spaces whenever it makes the statements easier to read. For example blanks positioned before and after the assignment sign, and after C++ keywords, make the following statements easier to read.

```
fXDifference = fx2-fx1; // difference in the x coordinates
// update max value
if (nV1 > nV2) n.MaxValue = nV1;
```

A blank line before and after a body of statements makes the body easier to read and understand.

```
// initialize all parameters to default values
Initialize ();

// now draw all lines
for (int i=1; i <= nLines; i++)
{
    GetLine (i);
    DrawLine (i);
}

// put point labels
ShowPointLabels ();
```

Programming Style Tip 4.4: Indentation and Braces

Indentation of the statements makes the program easier to follow. In the following statements, indentation is used to distinguish the different **if-else** block of statements, as well the statements belonging to the **for** loops. The block of statements have matching braces { .. }. In addition, the use of braces makes the **for** loop easier to read, though it is not necessary for correct program execution.

```
if (bComputeAverage)
{
    for (fAvg = 0.0f, int i = 1; i <= nPoints; i++)
    {
        fAvg += fValues(i);
    }
    fAvg /= float(nPoints);
}
else
{
    for (fSD = 0.0f, int i = 1; i <= nPoints; i++)
    {
        fSD += fValues(i)*fValues(i);
    }
    fSD = float(sqrt(fSD));
}
```

Programming Style Tip 4.5: Simple Statements

This is somewhat subjective because what is simple to an experienced programmer may be difficult to follow for the beginner programmer. However, we will develop programs that are easy to read and follow. For example, we can take a complex statement and break it into several easy to understand simple statements. The sequence of statements

```
float fOI, fII; // to store moment of inertia
fOI = (pow(wH+2.0f*fT, 3.0) * fW)/12.0f; // outer moment of inertia
fII = 2.0f*(pow(wH, 3.0) * (0.5f*(fW-wT)))/12.0f; // inner MOI
m_fSyy = (fOI - fII)/(0.5f*wH+fT); // section modulus
```

is easier to read than

```
m_fSyy = ((pow(wH+2.0f*fT, 3.0) * fW)/12.0f - 2.0f*(pow(wH, 3.0) *
(0.5f*(fW-wT)))/12.0f) /(0.5f*wH+fT);
```

Programming Style Tip 4.6: Naming iterator variables

Use **i**, **j**, **k** etc. as the variables to denote iterators. For example,

```
for (int i=1; i <= nLimits; i++)
{
    ...
}
```

Programming Style Tip 4.7: Modular Program Development 1

Whenever possible, we will develop program components (functions and classes) that can be reused. As a further refinement, template functions and classes will be developed and used whenever appropriate.

Programming Style Tip 4.8: Modular Program Development 2

We will store the modules in separate files. This will make the task of program development and maintenance easier.

Programming Style Tip 4.9: Use std::string to store character strings

Though we have not learnt about classes – how to define and use them, whenever possible, we will use the `string` class to store a string of characters. We will see a formal introduction to the standard `string` class in Chapter 7.

Programming Style Tip 4.10: Create easy to read tabular output

As we have seen with the simple and moderately complex examples so far in the book, it is much easier for one to digest the information if presented in a visually attractive form. Tabular output is one such form (see Example Programs 3.2.5). Graphical output is another attractive form and we will see more about this in Chapter 19.

Programming Style Tip 4.11: Preprocessor directives and macros

We strongly discourage the use of both manifest constants and macros. For example, instead of defining a constant as

```
#define PI 3.1415926
```

define the constant instead as

```
const float PI=3.1415926f;
```

Not only will the above declaration generate compiler errors if an attempt is made to modify the constant but will also allow debugger access to the constant.

Macros are cumbersome to maintain and can lead to subtle errors. We will see that using templates (Chapter 9) provides a much better alternative.

Programming Style Tip 4.12: Global variables

Once again, we will strongly discourage the use of global variables through the use of `extern` qualifier. Once we understand how to define and use classes, we will see how to organize the program to minimize, if not eliminate, the need for global variables.

Programming Style Tip 4.13: Templates

We will strongly urge the use of templates whenever appropriate. Templates not only reduce the size of the source statements in a program and make program maintenance easier, they make software reuse through the development of libraries possible. As we have mentioned before, one of the strengths of C++ is the ease with which libraries have developed and made available to programmers. The Standard Templates Library (STL) is one such example.

EXERCISES

Most of the problems below involve the development of one or more functions. In each case develop (a) a plan to test the function(s), and (b) implement the plan in a **main** program. The functions should not use **cin** or **cout** unless specified. Put the main program in a separate file and the function(s) in separate files.

Appetizers

Problem 4.1

Write a function **IsOdd** to determine whether an integer **n** is an odd number as

```
bool IsOdd (int n);
```

Problem 4.2

The thermal efficiency, e of a heat engine is defined as the ratio of the net work done to the thermal energy absorbed at the highest temperature during one cycle as

$$e = 1 - \frac{Q_c}{Q_h}$$

where Q_h is the amount of heat absorbed by the engine and Q_c is the amount of heat given up. The function prototype is given as follows.

```
float Efficiency (float fHeatAbsorbed, float fHeatLost);
```

Problem 4.3

The half-life of radon is 3.8 days. Write a function to compute the concentration of radon given the initial concentration (in *mol/L*) and the elapsed time (in days). The function prototype is given as

```
float RadonConc (float fInitialConc, int nDays);
```

Problem 4.4

The Arrhenius Equation relates the rate at which a reaction proceeds with its temperature and is given as

$$k = A e^{(-E_a/RT)}$$

where k is rate coefficient, A is a constant, E_a is the activation energy, R is the universal gas constant, T is the temperature in degrees Kelvin. Write a function to compute the rate coefficient

(stored in a vector) for a given number of values of temperatures (stored in a vector). `nPoints` is the number of temperature values.

```
void Arrhenius (float fA, float fEa, const float fVT[], float fVK[],  
int nPoints);
```

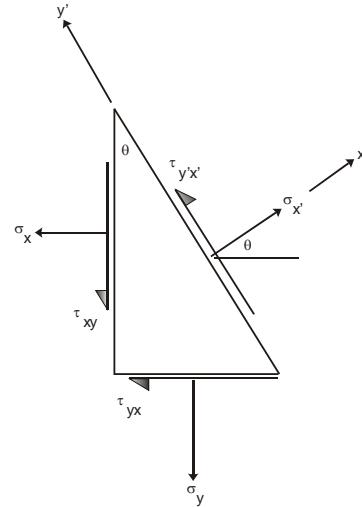
Problem 4.5

Write a function to compute the state of stress $(\sigma_{x'}, \tau_{x'y'})$ on a plane given $(\sigma_x, \sigma_y, \tau_{xy})$ and θ in degrees.

```
void StressTransform (float fSigx, float fSigy, float fTauxy, float& fSigxP,  
float& fTauPyP, float fTheta);
```

$$\sigma_{x'} = \frac{\sigma_x + \sigma_y}{2} + \frac{\sigma_x - \sigma_y}{2} \cos 2\theta + \tau_{xy} \sin 2\theta$$

$$\tau_{x'y'} = -\frac{\sigma_x - \sigma_y}{2} \sin 2\theta + \tau_{xy} \cos 2\theta$$



Main Course

Problem 4.6

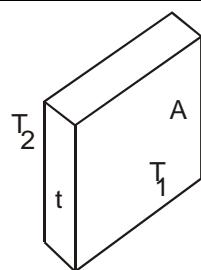
Develop a library of functions to compute the surface area and volume of the following three-dimensional objects – (1) Cube, (2) Tetrahedron, (3) Right pyramid, (4) Right circular cylinder, (5) Right circular cone, (6) Sphere. The function prototypes are presented below.

```
void CubeProp (float& fSurfArea, float& fVolume, float fSide);  
void TetrahedronProp (float& fSurfArea, float& fVolume, const float fVX[4],  
const float fVY[4], const float fVZ[4]);  
void PyramidProp (float& fSurfArea, float& fVolume, float fSide,  
float fHeight);  
void CylinderProp (float& fSurfArea, float& fVolume, float fRadius,  
float fHeight);  
void ConeProp (float& fSurfArea, float& fVolume, float fRadius,  
float fHeight);  
void SphereProp (float& fSurfArea, float& fVolume, float fRadius);
```

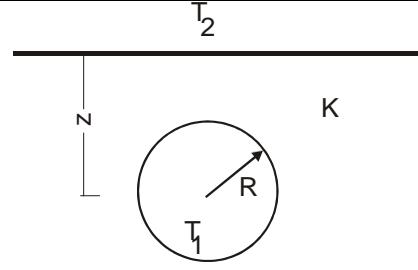
Problem 4.7

Conduction of heat through different shaped solid objects. The total heat transfer rate Q through the body is related to the temperature difference ΔT and a quantity called the conduction shape factor S by $Q = Sk\Delta T = Sk(T_1 - T_2)$ where k is the thermal conductivity of the body and T_1 and T_2 are the boundary surface temperatures across which heat flow takes place. The shape factor S is related to the thermal resistance of the body and is given as shown in the following table.

Shape	S
Slab of thickness t and cross-sectional area of heat flow A (see Fig. (a))	A/t
Long hollow cylinder of length L , inner radius r_1 at temperature T_1 , outer radius r_2 at temperature T_2	$\frac{2\pi L}{\ln(r_2/r_1)}$
A sphere of radius R maintained at temperature T_1 placed in a semi-infinite medium at a distance z from a surface maintained at temperature T_2 (see Fig. (b))	$\frac{4\pi R}{1 - R/(2z)}$
A sphere of radius R maintained at temperature T_1 placed in a semi-infinite medium maintained at temperature T_2 and placed at a distance z from an insulated surface	$\frac{4\pi R}{1 + R/(2z)}$
A cylinder of radius R and length L maintained at temperature T_1 placed horizontally in a semi-infinite medium at a distance z from a surface maintained at temperature T_2	$\frac{2\pi L}{\cosh^{-1}(z/R)}$
Circular hole of radius R centered in a square solid of side a and length L	$\frac{2\pi L}{\ln(0.54a/R)}$



(a)

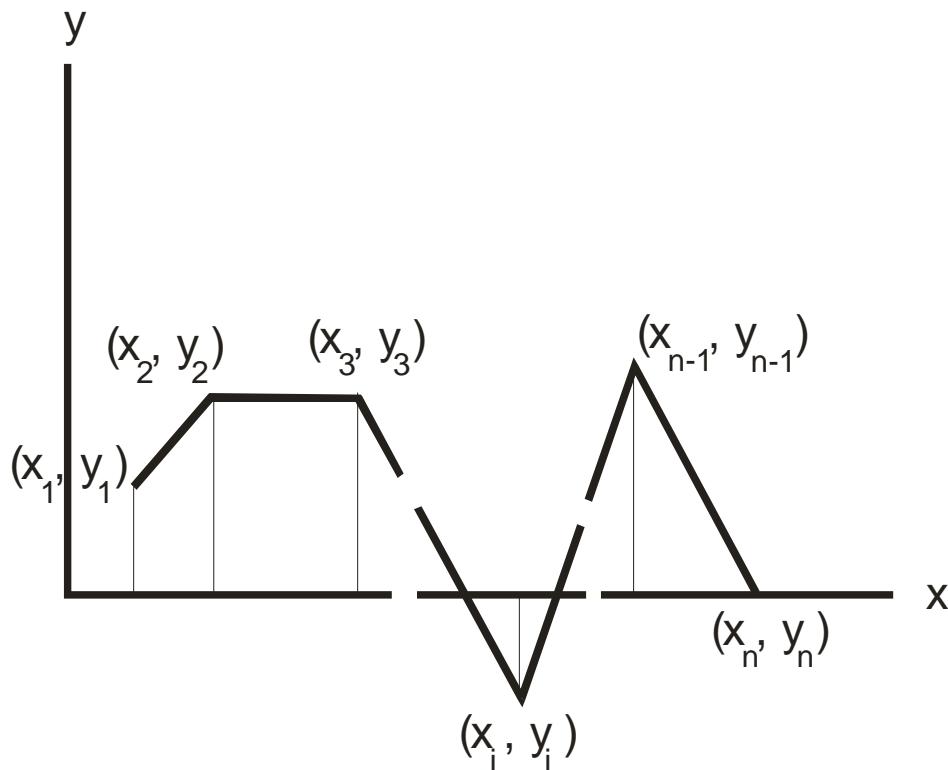


(b)

Develop a program to obtain the values of different parameters and compute the value of Q . Construct a separate function for each of the shapes shown in the table.

Problem 4.8

Write a function to compute the area under a curve that is approximated as straight lines between adjacent points as shown in the figure below. With this scheme, the shape under the curve between adjacent points can either be a trapezoid or a rectangle or a triangle (note: area can be positive or negative or zero).



The prototype of this function is as follows.

```
float AreaUnderCurve (const float fVX[], const float fVY[], int nPoints,
                      int& nTriangles, int& nRectangles, int& nTrapezoids);
```

The inputs to the function are the vector of x and y values of the points. The output is the return value that is the area under the curve and the last three arguments – the number of triangles, rectangles and trapezoids detected during the computation of the area. Develop three other functions

```
float AreaRightTriangle (float fHeight, float fBase);
float AreaRectangle (float fHeight, float fBase);
float AreaTrapezoid (float fBase, float fHeightLeft, float fHeightRight);
```

to compute the area of a right triangle, rectangle and a trapezoid. Call these functions from the `AreaUnderCurve` function.

Problem 4.9

Fibonacci numbers are defined as the sequence of the following integers $0, 1, 1, 2, 3, 5, 8, \dots$. In other words, $F_1 = 0, F_2 = 1$ and $F_i = F_{i-1} + F_{i-2}, i = 2, 3, \dots$. Write two functions to compute the Fibonacci numbers with the first using iterations and the other using recursion.

Problem 4.10

Develop a library of functions to operate on points in the (x, y, z) space that are stored as a double vector of length 3. The following functions are needed.

Distance: The straight line distance between points 1 and 2.

DistanceFromOrigin: The straight line distance between the point and the origin of the coordinate system.

UnitVector: Unit vector between points 1 and 2.

DistanceFromLine: Shortest distance from the point to the straight line connecting points 1 and 2.

The function prototypes are given below.

```
double Distance (const double dv1[3], const double dv2[3]);
double DistanceFromOrigin (const double dvp[3]);
void UnitVector (const double dv1[3], const double dv2[3],
                  double dvUnitV[3]);
double DistanceFromLine (const double dvp[3], const double dv1[3],
                        const double dv2[3]);
```

Once you have tested the functions, convert them to template functions so that they can be used with either `float` or `double` data types.

C++ Concepts

Problem 4.11

Enhance the library of statistical functions shown in Example Program 4.4.1. Develop these functions as template functions. The functions should compute the following statistical measures – (1) Arithmetic mean, (2) Geometric Mean, (3) Median, (4) Mean Deviation, (5) Standard Deviation, (6) Variance, and (7) Covariance.

Problem 4.12 (See Problems 1.9 and 1.10)

A number of engineering problems are solved using heuristics. One can loosely define heuristics as the employment of the solution techniques that are based on experience rather than on a rigorous theory. The use of rule-based procedures is an example of heuristics. We all use heuristics on a regular basis

especially when playing games. Write a computer program for playing tic-tac-toe. Start with an initial screen that looks similar to the grids shown below.

1	2	3
4	5	6
7	8	9

User plays first

1	2	3
4	5	6
X	8	9

Computer plays first

Chapter 5

Numerical Analysis: Introduction

“Learning without thought is useless, thought without learning is dangerous.” Confucius .

“You can use an eraser on the drafting table or a sledge hammer on the construction site.” Frank Lloyd Wright

“If a little knowledge is dangerous, where is the man who has so much as to be out of danger?”
Thomas H. Huxley

In the preceding chapters we saw how to write simple yet useful programs. As we have repeatedly seen before, good computer programs can be used to among other things, automate mundane, repeating tasks and make them as error-free as possible. In this chapter we will start to look at the basics of numerical analysis. We will see how integer and floating point numbers are represented and stored. We will look at accuracy, sources of errors and how best to deal with computer arithmetic.

Objectives

- To understand the basics of numerical analysis starting with numerical representation.
- To understand what is meant by numerical approximation and numerical errors.
- To understand Taylor series expansion and function approximation.

5.1 Approximations and Errors

In this introductory section we will see how numbers are represented in computers, how computer arithmetic takes place, and what are the sources of numerical errors?

Significant Digits

Scientific notation is the representation of floating point numbers as

$$\pm a.bcd \times 10^{\pm xyz}$$

where a, b, c, d, x, y, z are integers between 0 and 9. The significant digits in a number represent the digits that are known to be correct or with confidence. For example, 0.0054 and 0.054 are both known to have two significant digits. In these examples, the zeros appearing after the decimal (leading zeros) help to locate the first nonzero entry and hence the zeros are not important. On the other hand, how many significant digits are there in 0.0540 or in 5400 where we have trailing zeros? A better way of answering that question is to write the numbers in scientific notation as 5.40×10^{-2} and 5.4×10^3 . With this notation, 0.0540 has three significant digits and 5400 has two significant digits. If on the other hand, 5400 is expressed as 5.400×10^3 , then the number has 4 significant digits.

A number \hat{x} is an approximation to its true value x to s significant digits if the following inequality is satisfied by the largest positive integer s

$$\left| \frac{x - \hat{x}}{x} \right| < \frac{10^{-s}}{2} \quad (5.1.1)$$

One could ask “Find x_a to approximate 500 to 3 significant digits.” Using the above definition, we can express the solution as

$$\left| \frac{500 - x_a}{500} \right| < \frac{10^{-3}}{2}$$

from which $499.75 < x_a < 500.25$.

Numerical Representation

We as human beings are taught to recognize and manipulate numbers using the decimal system where the digits go from 0 to 9. For example,

$$(4096)_{10} = 6 \times 10^0 + 9 \times 10^1 + 0 \times 10^2 + 4 \times 10^3.$$

Numbers in computer systems are represented as binary (or sometimes octal or hexadecimal) numbers. In the binary system, the digits are either 0 or 1. For example,

$$(10011)_2 = 1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 0 \times 2^3 + 1 \times 2^4 = (19)_{10}$$

In computer terminology, a **bit** is the smallest unit of storage. In other words, a bit can either store a 0 or a 1 value. Bits are grouped together to form **bytes**. For example, 8 bits form a byte. Bytes can then be grouped together to form **words**. On most 32-bit hardware systems such as those manufactured by Intel and AMD, 4 bytes (or 32 bits) form an integer word and a single-precision word. 8 bytes (or 64 bits) form a double-precision word.

Consider a hypothetical computer system where integers are stored in 4 bits. Since integers are signed meaning that they can be either negative or positive, one of these bits is used to store the sign (Fig. 5.1.1).

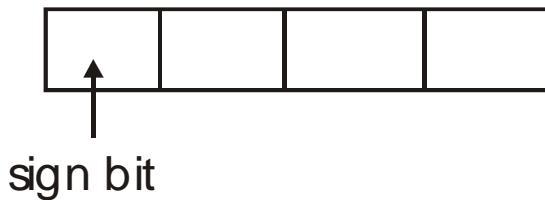


Fig. 5.1.1 Bit representation for a 4-bit integer storage

A 1 in the sign bit usually signifies a negative number. The other three bits are then used to store the value of the integer. The largest value occurs when all the three bits are 1's. In other words the largest value than can be stored is $(111)_2 = 1 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 = (7)_{10}$. Hence the range of integer numbers, n that can be represented in a 4-bit representation is $-7 \leq n \leq 7$. Or, the range of (decimal) numbers, n that can be represented in a p -bit representation is $-(2^{p-1} - 1) \leq n \leq (2^{p-1} - 1)$. Going back to Section 2.2, we can now see why a *short* number using 16 bits can be used to store values between $-(2^{16-1} - 1) \leq n \leq (2^{16-1} - 1) \rightarrow -32767 \leq n \leq +32767$.

When a floating point number is represented in the decimal system, we can continue to think of the number as we did with integers. For example,

$$(4.203)_{10} = 4 \times 10^0 + 2 \times 10^{-1} + 0 \times 10^{-2} + 3 \times 10^{-3}$$

Similarly, for the binary representation of floating point numbers, we have the following example.

$$(1.101)_2 = 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = (1.625)_{10}$$

The computer representation of a floating-point number is a little more complex. Typically, a floating point number has three components (Fig. 5.1.2).

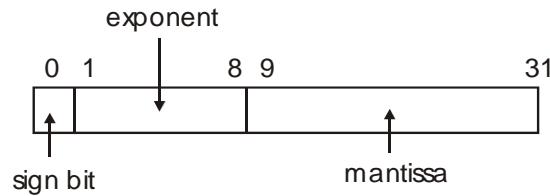


Fig. 5.1.2 IEEE representation of single precision floating point number

In other words, the number x is stored as a binary approximation as

$$x \approx \pm q \times 2^n \quad (5.1.2)$$

where q is the mantissa and n is the exponent. The IEEE single precision floating point standard representation requires a 32 bit word¹, which may be represented as numbered from 0 to 31, left to right. The first bit is the sign bit, S, the next eight bits are the exponent bits, 'E', and the final 23 bits are the fraction 'F':

S	EEEEEEEEE	FFFFFFFFFFFFFFFFFFFFFFFF		
0	1	8	9	31

The value V represented by the word may be determined as follows:

- If E=255 and F is nonzero, then V=NaN ("Not a number")
 - If E=255 and F is zero and S is 1, then V=-Infinity
 - If E=255 and F is zero and S is 0, then V=Infinity
 - If $0 < E < 255$ then $V = (-1)^S \times 2^{E-127} \times (1.F)$ where "1.F" is intended to represent the binary number created by prefixing F with an implicit leading 1 and a binary point.
 - If E=0 and F is nonzero, then $V = (-1)^S \times 2^{-126} \times (0.F)$ These are "unnormalized" values.
 - If E=0 and F is zero and S is 1, then V=-0
 - If E=0 and F is zero and S is 0, then V=0

The exponent can either be negative or positive. A bias is subtracted from the exponent in order to get the actual exponent. This bias value is 127 for single-precision floats. As an example, an exponent (E) value of 134 means that the actual exponent is $(134 - 127)$, or 7.

¹ ANSI/IEEE Standard 754-1985,
Standard for Binary Floating Point Arithmetic

The mantissa represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits. In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in normalized form. This basically puts the radix point after the first non-zero digit. Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa has effectively 24 bits of resolution, by way of 23 fraction bits.

For example,

$$1\ 10000001\ 10100000000000000000000 = -1 * 2^{**(129-127)} * (1.101)_2 = -(6.5)_{10}$$

Since the number of bits for the exponent is 8, the approximate range of numbers that can be represented is $\pm 2^{(2^8-1-127)} = \pm 3.4(10^{38})$. There are five distinct numerical ranges that single-precision floating-point numbers are not able to represent.

(1) Negative overflow: Negative numbers less than $-(2 - 2^{-23}) \times 2^{127}$.

(2) Negative underflow: Negative numbers greater than -2^{-149} .

(3) Zero (see below).

(4) Positive underflow: Positive numbers less than 2^{149} .

(5) Positive overflow: Positive numbers greater than $(2 - 2^{-23}) \times 2^{127}$.

Overflow means that values have grown too large for the representation, much in the same way that you can overflow integers. Underflow is a less serious problem because it just denotes a loss of precision, which is guaranteed to be closely approximated by zero.

The procedure to store and interpret double precision numbers is very similar to single precision numbers (Fig. 5.1.3).

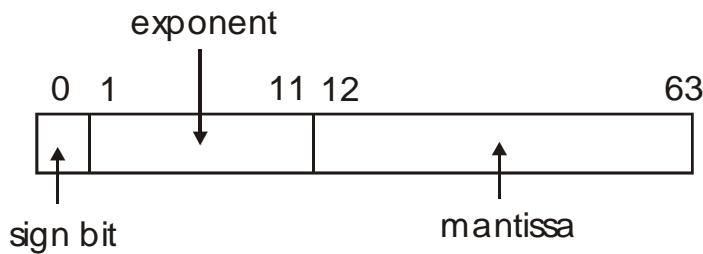


Fig. 5.1.3 IEEE representation of double precision floating point number

There are some special values that one should be aware of. IEEE reserves exponent field values of all 0s and all 1s to denote special values in the floating-point scheme.

Zero

As mentioned before, zero is not directly representable in the straight format, due to the assumption of a leading 1 (we'd need to specify a true zero mantissa to yield a value of zero). Zero is a special value denoted with an exponent field of zero and a fraction field of zero. Note that -0 and +0 are distinct values, though they both compare as equal.

Unnormalized

If the exponent is all 0s, but the fraction is non-zero (else it would be interpreted as zero), then the value is a denormalized number, which does not have an assumed leading 1 before the binary point. Thus, this represents a number $(-1)^s \times 0.f \times 2^{-126}$, where s is the sign bit and f is the fraction. For double precision, denormalized numbers are of the form $(-1)^s \times 0.f \times 2^{-1022}$. From this you can interpret zero as a special type of unnormalized or denormalized number.

Infinity

The values $+\infty$ and $-\infty$ are denoted with an exponent of all 1s and a fraction of all 0s. The sign bit distinguishes between negative infinity and positive infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations. Operations with infinite values are well defined in IEEE floating point.

Not A Number

The value NaN (Not a Number) is used to represent a value that does not represent a real number. NaN's are represented by a bit pattern with an exponent of all 1s and a non-zero fraction. There are two categories of NaN: QNaN (Quiet NaN) and SNaN (Signalling NaN).

A QNaN is a NaN with the most significant fraction bit set. QNaN's propagate freely through most arithmetic operations. These values pop out of an operation when the result is not mathematically defined.

An SNaN is a NaN with the most significant fraction bit clear. It is used to signal an exception when used in operations. SNaN's can be handy to assign to uninitialized variables to trap premature usage.

Semantically, QNaN's denote indeterminate operations, while SNaN's denote invalid operations.

Converting a Decimal Number to a Non-Decimal Number (Base b)

So as to generalize the procedure for both integers and floating point numbers, we will split the given number into its integral and fractional parts. Note that an integer has no fractional part.

Integral Part

- (1) Divide the integral part by the base b . This yields a quotient and a remainder. The remainder is the rightmost digit of the integral part of the new number.

- (2) Divide the quotient again by b . The remainder is the next digit of the integral part.
- (3) Repeat step (2) until the quotient is zero. The (last) remainder is the leftmost digit of the new number.

Fractional Part

- (1) Multiply the fractional part of the decimal number by base b . The integral part of the product constitutes the leftmost digit of the fractional part of the new number.
- (2) Multiply the fractional part of the product by base b . The integral part of the product constitutes the next digit of the fractional part of the new number.
- (3) Repeat step (2) until a zero fractional part or a duplicate fractional part occurs. The integer part of the (last) product is the rightmost digit of the fractional part of the new number. A duplicate fractional part is an indication that the digit (or sequence) is a repeating one.

Example 5.1

Problem Statement: Represent each of the following decimal numbers as a binary numbers.

- (a) 12 (b) -24 (c) -1.45

Solution: For each of the numbers we present a table showing the calculations.

(a) $(12)_{10} = (1100)_2$

Division	(Quotient, Remainder)	Binary Number
12/2	(6, 0)	0
6/2	(3, 0)	00
3/2	(1, 1)	100
1/2	(0, 1)	1100

(b) $(-24)_{10} = (-11000)_2$ ²

² Negative numbers are usually represented as 2's complement. See Problem 5.9.

Division	(Quotient, Remainder)	Binary Number
24/2	(12, 0)	0
12/2	(6, 0)	00
6/2	(3, 0)	000
3/2	(1, 1)	1000
1/2	(0, 1)	11000

$$(c) (-1.45)_{10} = (-1.0\overline{1100}...)_2$$

Integral Part

Division	(Quotient, Remainder)	Binary Number
1/2	(0, 1)	1

Fractional Part

Multiplication	(Product, Integral Part)	Binary Number
0.45 x 2	(0.90, 0)	0
0.90 x 2	(1.80, 1)	01
0.80 x 2	(1.60, 1)	011
0.60 x 2	(1.20, 1)	0111
0.20 x 2	(0.40, 0)	01110
0.40 x 2	(0.80, 0)	011100
0.80 x 2	(1.60, 1)	011100...

As we can see from the last row in the table, the pattern begins to repeat itself; hence the calculations are terminated. It should be noted with this simple example, a number that can be represented exactly as a decimal number may not be represented exactly as a binary number with a finite number of bits.

Types of Errors

Before we discuss the various types of errors that can result from computer arithmetic, let us first define two very important terms – absolute error and relative error. Let x_t be the true value of a quantity whose computed approximate value is denoted as x_a . The absolute error, E_{abs} is then given as

$$E_{abs} = |x_t - x_a| \quad (5.1.3)$$

and the relative error, E_{rel} is defined as

$$E_{rel} = \frac{|x_t - x_a|}{|x_t|} \quad (5.1.4)$$

While the signs are sometimes useful, usually we are more concerned with the magnitude of the error. Hence the absolute values are used in both the error definitions. Both these error measures tell us something about how accurate the approximate value is.

Example 5.2

Problem Statement: (a) The weight of a certain object is 15.0 N. A store clerk weighs the object and reports the weight as 15.5 N. What are the absolute and relative errors in the clerk's measurement?

(b) A student astronomer using a telescope estimates the distance to a celestial object as 15,500,000 miles. It is known that the celestial object is in fact 15,000,000 miles away. What are the absolute and relative errors in the student's measurement?

Solution:

(a) From the problem data we have $x_t = 15.0$ and $x_a = 15.5$. Using Eqn. (5.1.3) we have

$$E_{abs} = |x_t - x_a| = |15.0 - 15.5| = 0.5N$$

Using Eqn. (5.1.4) we have

$$E_{rel} = \frac{|x_t - x_a|}{|x_t|} = \frac{|15.0 - 15.5|}{|15.0|} = \frac{0.5}{15.0} = 0.0333$$

(b) From the problem data we have $x_t = 15000000$ and $x_a = 15500000$. Using Eqn. (5.1.3) we have

$$E_{abs} = |x_t - x_a| = |15000000 - 15500000| = 500000 \text{ miles}$$

Using Eqn. (5.1.4) we have

$$E_{rel} = \frac{|x_t - x_a|}{|x_t|} = \frac{|15000000 - 15500000|}{|15000000|} = \frac{500000}{15000000} = 0.0333$$

This simple example shows why both error measures are necessary to draw conclusions. The absolute error in the weight estimate is 0.5 and 500000 in the distance estimate. One may incorrectly conclude that the weight measurement is more accurate. However, when we compare the relative errors, both errors are exactly the same – 3.33%.

Now we will look at errors resulting from computer arithmetic – round-off errors and truncation errors.

Round-Off Errors

Round-off errors occur because a fixed number of bits are used to represent numbers. For example, the fraction $1/3$ cannot be represented exactly as a decimal when a fixed number of bits are available. Similarly, as we saw with Example 5.1(c), all decimal numbers cannot be represented exactly using finite number of bits as binary numbers. Round-off errors can occur at all stages of numerical computations and the cumulative effect can be large.

A floating-point number can be represented as

$$x = x_a \times 10^n + x_e \times 10^{n-d} = \text{approx } x + \text{error}$$

where d is the number of digits available for the mantissa. For example, consider a computer representation of floating-point numbers with a fixed word length of 6 digits. Suppose we want to represent the number 199.05678 using such a representation. The number can be represented as

$$\begin{aligned} 199.05678 &= 0.19905678 \times 10^3 \\ &= (0.199056 + 0.78 \times 10^{-6}) \times 10^3 \end{aligned}$$

In this example, the digits after 6 are dropped (chopped off). When this procedure is used, we can bound the error as

$$\text{Error} \leq 10^{n-d} \quad (5.1.5)$$

Hence the truncation error is $x_e \times 10^{n-d} = 0.00078$. On the other hand, if symmetric roundoff is used, the error is usually smaller. In symmetric roundoff, the last significant digit is rounded up by 1 if the first dropped digit is larger than or equal to 5. Hence, going back to our example, we have

$$199.05678 = 0.19905678 \times 10^3$$

$$\approx 0.199057 \times 10^3$$

Note that the first dropped digit is 7 and the last retained digit is 6 that is rounded up to 7. We can bound the error for symmetric roundoff as

$$\text{Error} = x_e \times 10^{n-d} \quad x_e < 0.5$$

$$\text{Error} = (x_e - 1) \times 10^{n-d} \quad x_e \geq 0.5$$

Hence,

$$\text{Error} \leq 0.5 \times 10^{n-d} \quad (5.1.6)$$

which when compared to Eqn. (5.1.3) is at the worst case half as big.

Truncation Errors

Truncation errors occur because when an approximation is used in the place of an exact representation. For example, in the evaluation of a transcendental function involving an infinite series, a truncated series is used in the numerical evaluation of its value. Consider the following series to evaluate the sine value.

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \dots \quad (5.1.7)$$

Since all the terms in the infinite series cannot be evaluated, the series is truncated after a certain number of terms. Hence, the term *truncation error*.

Consider the case where $\sin 0.5$ is evaluated. When the first three terms are used, we have

$$\sin(0.5) = 0.5 - \frac{0.5^3}{3!} + \frac{0.5^5}{5!} = 0.47942708$$

and when the first four terms are used we have

$$\sin(0.5) = 0.5 - \frac{0.5^3}{3!} + \frac{0.5^5}{5!} - \frac{0.5^7}{7!} = 0.47942553$$

A more accurate value computed using extended precision is 0.47942553860420300027328793521557.

Machine Epsilon or Precision

Another important quantity is known as machine epsilon. Machine epsilon, ε is the upper bound on the relative error that occurs when a nonzero real number, x is represented as a floating point number x_a . In other words

$$\left| \frac{x - x_a}{x} \right| \leq \varepsilon \quad (5.1.8)$$

We can customize the above expression for computer systems that use base b with d -digit mantissa. When truncation is used

$$\varepsilon = b^{-d+1} \quad (5.1.9a)$$

and when symmetric rounding is carried out

$$\varepsilon = 0.5 \times b^{-d+1} \quad (5.1.9b)$$

There is another way of defining machine epsilon (also known as unit roundoff). Let x_a be the smallest number representable in the machine arithmetic that is greater than 1 (in the machine). The machine epsilon is then defined as

$$\varepsilon = x_a - 1 \quad (5.1.10)$$

We will use this definition to estimate the machine epsilon and show that Eqns. (5.1.9) and (5.1.10) are equivalent.

Example Program 5.1.1 Computing machine precision

In the example shown below, the machine epsilon is estimated using Eqn. (5.1.10). In other words, we will add a floating point number to 1.0 and check to see if the sum is 1.0. If not, we will divide the number by 2 until the number becomes so small that adding it to 1.0 will yield a result of 1.0.

main.cpp

```

1 #include <iostream>
2 #include <iomanip>
3 #include <cmath>
4
5 int main ()
6 {
7     float fEP=1.0f, fx;
```

```

8      int n = 0;
9      std::cout << std::setprecision(15) << std::scientific;
10
11     for (;;)
12     {
13         fX = 1.0f + fEP;
14         std::cout << std::right
15             << std::setw(20) << "fEP = " << fEP << ". n = "
16             << std::left << std::setw(20) << n << "\n";
17         if (fX == 1.0f)
18             break;
19         fEP = fEP/2.0f;
20         ++n;
21     }
22
23     std::cout << "\n\nEstimate of machine epsilon : " << fEP << "\n\n";
24
25     return 0;
26 }
```

In line 7, the floating-point number (that is added to 1.0) is itself initialized to 1.0. In line 19, this value is halved. The process is repeated until adding the number to 1.0 (line 13) does not change the result from 1.0.

Computer Arithmetic

As we have seen earlier, floating point numbers can be represented to a finite precision. In the following examples, we will illustrate errors resulting from computer arithmetic.

Example Program 5.1.2 Effect of precision

In this example we will examine the effect of finite precision that exists in any computer system. The problem is to compute the sum of the series implied in $\sum_{i=1}^n \frac{1}{i^2}$. The accuracy of the evaluation should increase with increasing values of n . Note that the analytical result is $\pi^2/6$.

main.cpp

```

1 #include <iostream>
2 #include <iomanip>
3 #include <cmath>
4
5 int main ()
6 {
7     int N;
8     std::cout << "Input value for N: ";
9     std::cin >> N;
10
11    std::cout << std::setprecision(15) << std::scientific;
12
13    // -----
14    // single precision
```

```

15      // -----
16      std::cout << "Single precision results ....\n";
17      const float PISP = 3.1415926f;
18      float fSum = 0.0f;
19      // process 1
20      for (int i=1; i <= N; i++)
21          fSum += 1.0f/(static_cast<float>(i)*static_cast<float>(i));
22      std::cout << "Process 1: Sum is " << std::setw(20) << fSum << '\n';
23
24      // process 2
25      fSum = 0.0f;
26      for (int i=N; i >= 1; i--)
27          fSum += 1.0f/(static_cast<float>(i)*static_cast<float>(i));
28      std::cout << "Process 2: Sum is " << std::setw(20) << fSum << '\n';
29      std::cout << "Exact answer is " << PISP*PISP/6.0f << "\n\n";
30
31      // -----
32      // double precision
33      // -----
34      std::cout << "Double precision results ....\n";
35      const double PIDP = 3.14159265358979;
36      double dSum = 0.0;
37      // process 1
38      for (int i=1; i <= N; i++)
39          dSum += 1.0/(static_cast<double>(i)*static_cast<double>(i));
40      std::cout << "Process 1: Sum is " << std::setw(20) << dSum << '\n';
41
42      // process 2
43      dSum = 0.0;
44      for (int i=N; i >= 1; i--)
45          dSum += 1.0/(static_cast<double>(i)*static_cast<double>(i));
46      std::cout << "Process 2: Sum is " << std::setw(20) << dSum << '\n';
47      std::cout << "Exact answer is " << PIDP*PIDP/6.0 << '\n';
48
49      return 0;
50  }

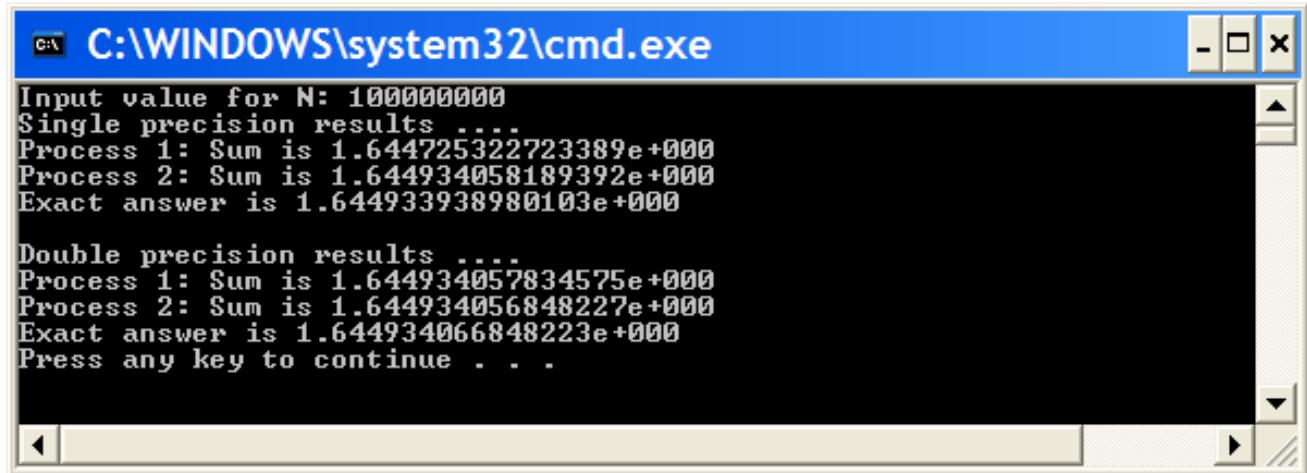
```

The program computes the sum of the infinite series using both single and double precision after having obtained the value of n from the user. A sample output is shown in Fig. 5.1.4 for a relatively large value of n - 100 million. For both single and double precision, the sum of the series is computed two different ways as follows

$$S_{forward} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \dots + \frac{1}{n^2} \quad (5.1.11a)$$

$$S_{reverse} = \frac{1}{n^2} + \frac{1}{(n-1)^2} + \frac{1}{(n-2)^2} + \dots + \frac{1}{2^2} + \frac{1}{1^2} \quad (5.1.11b)$$

One would expect that there would be no difference between the two procedures. However, the sample output shows that $S_{reverse}$ is more accurate. In Eqn. (5.1.11b), the sum is obtained by adding from the smallest to the largest number.



The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window contains the following text:

```

Input value for N: 1000000000
Single precision results ...
Process 1: Sum is 1.644725322723389e+000
Process 2: Sum is 1.644934058189392e+000
Exact answer is 1.644933938980103e+000

Double precision results ....
Process 1: Sum is 1.644934057834575e+000
Process 2: Sum is 1.644934056848227e+000
Exact answer is 1.644934066848223e+000
Press any key to continue . .

```

Fig. 5.1.4 Sample output generated by using n as 100 million

Atkinson [1978] shows that if truncation is used rather than rounding, and if all numbers are positive, the strategy of adding from smallest to largest minimizes the effect of these chopping errors.

Example Program 5.1.3 Numerical errors

As we discussed earlier, there are different types of numerical errors that can take place in a computer program. In this example, we will look at some of them. The source code artificially generates these errors – divide by zero, illegal operation, floating point overflow and underflow, and an operation that yields a NaN.

main.cpp

```

1 #include <iostream>
2 #include <cmath>
3 #include <cfloat>
4
5 int main ()
6 {
7     float fX = -33.56f;
8     float fY = 0.0f;
9
10    float fA = fX/fY;           // divide by zero
11    std::cout << fX << "/" << fY << " = " << fA << "\n";
12
13    float fB = sqrt(fX);       // illegal operation
14    std::cout << "sqrt(" << fX << ") = " << fB << "\n";
15
16    double dU = 4.5e155, dV = 4.6e185;
17    double dC = dU * dV;        // overflow
18    std::cout << dU << " * " << dV << " = " << dC << "\n";
19
20    double dQ = 4.5e-135, dR = 4.6e-195;
21    double dD = dQ * dR;        // underflow
22    std::cout << dQ << " * " << dR << " = " << dD << "\n";

```

```

23
24     float fE = fY/fY;           // NaN (not a number)
25     if (_isnan(fE))
26         std::cout << fY << " / " << fY << " is not a number.\n";
27     Else
28         std::cout << fY << " / " << fY << " = " << fE << "\n";
29
30     return 0;
31 }

```

Fig. 5.1.5 shows the output generated by the program. In line 25, the program uses Microsoft-specific function `_isnan` to check whether the result of the division (from line 24) stored in `fE` yields a valid number or not.

```

D:\WINWORD\BOOKS\OOP\PROGRAMS\Example5_1_3\Debug\Example5_1_3.exe
-33.56/0 = -1.#INF
sqrt(-33.56) = -1.#IND
4.5e+155 * 4.6e+185 = 1.#INF
4.5e-135 * 4.6e-195 = 0
0 / 0 is not a number.
Press any key to continue...

```

Fig. 5.1.5 Output generated by Microsoft C++ compiler

Finally we will look at what C++ provides us with respect to integer and floating point numbers.

Example Program 5.1.4 Numerical limits

This example program shows how to extract the numerical limits for integer and floating point numbers on a particular operating system and hardware.

main.cpp

```

1  #include <iostream>
2  #include <limits>          // if necessary use climits
3
4  int main ()
5  {
6      std::cout << " Machine epsilon for a float number is "
7          << std::numeric_limits<float>::epsilon() << '\n'
8          << "Machine epsilon for a double number is "
9          << std::numeric_limits<double>::epsilon() << '\n'
10         << "                                         Max int value is "
11         << std::numeric_limits<int>::max() << '\n'
12         << "                                         Max long value is "
13         << std::numeric_limits<long>::max() << '\n'
14         << "                                         Max float value is "
15         << std::numeric_limits<float>::max() << '\n'
16         << "                                         Max double value is "
17         << std::numeric_limits<double>::max() << '\n'
18         << "                                         Min int value is "

```

```

19         << std::numeric_limits<int>::min()      << '\n'
20         << "                                Min long value is "
21         << std::numeric_limits<long>::min()      << '\n'
22         << "                                Min float value is "
23         << std::numeric_limits<float>::max()     << '\n'
24         << "                                Min double value is "
25         << std::numeric_limits<double>::max()    << '\n'
26         << "      Machine radix or base for int is "
27         << std::numeric_limits<int>::radix      << '\n'
28         << "      Machine radix or base for float is "
29         << std::numeric_limits<float>::radix      << '\n';
30
31     return 0;
32 }

```

Since these constants are machine dependent, C++ provides a very convenient mechanism to find these machine dependent values. A sample output is shown in Fig. 5.1.6.

```

C:\WINDOWS\system32\cmd.exe
Machine epsilon for a float number is 1.19209e-007
Machine epsilon for a double number is 2.22045e-016
    Max int value is 2147483647
    Max long value is 2147483647
    Max float value is 3.40282e+038
    Max double value is 1.79769e+308
    Min int value is -2147483648
    Min long value is -2147483648
    Min float value is 3.40282e+038
    Min double value is 1.79769e+308
    Machine radix or base for int is 2
    Machine radix or base for float is 2
Press any key to continue . . .

```

Fig. 5.1.6 Machine-dependent values for Windows 7 using MSVS 2012 C++ compiler

5.2 Series Expansion

Having introduced the concept of errors including truncation error, we will now see one of the reasons why an understanding of errors is helpful. We will look at Taylor Series expansion that will in later chapters be used in problems such as root finding, numerical differentiation, minimization and maximization, ordinary differential equations etc. The basic motivation is to use polynomials to approximate continuous functions.

The Weierstrass Theorem

The theorem states that if $f(x)$ is a continuous function for $a \leq x \leq b$ and $\varepsilon > 0$, then there is a polynomial $p(x)$ for which

$$|f(x) - p(x)| \leq \varepsilon \quad a \leq x \leq b$$

Polynomials are attractive because computation of their derivatives and indefinite integrals is easy and the results are also polynomials.

Taylor Series Expansion

Taylor's Theorem states that if function $f(x)$ and its $(n+1)$ derivatives are continuous in the interval containing a , then the function can be approximated as a polynomial of the form

$$\begin{aligned} f(x) &= f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \dots \\ &\quad \frac{f^{(n)}(a)}{n!}(x-a)^n + R_n \end{aligned} \tag{5.2.1}$$

where the remainder is given by

$$R_n = \int_a^x \frac{(x-t)^n}{n!} f^{(n+1)}(t) dt \tag{5.2.2}$$

Eqn. (5.2.1) represents an infinite series and the theorem provides a mechanism for constructing different types of approximations (with finite number of terms) that can then be used in an effective manner to construct numerical solutions. Consider the following three approximations.

$$(a) \quad f(x) \approx f(a) \tag{5.2.3}$$

$$(b) \quad f(x) \approx f(a) + f'(a)(x-a) \tag{5.2.4}$$

$$(c) \quad f(x) \approx f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 \tag{5.2.5}$$

Eqn. (5.2.3) represents a zero-order approximation. If points x and a are sufficiently close to one other, then the approximation is reasonably good. Eqns. (5.2.4) and (5.2.5) represent the first and second-order approximations. In general, both provide better approximations than the zero-order approximation but at an additional cost of having to evaluate more terms. We can rewrite Eqn. (5.2.1) as

$$\begin{aligned} f(x_{i+1}) &= f(x_i) + f'(x_i)(x_{i+1} - x_i) + \frac{f''(x_i)}{2!}(x_{i+1} - x_i)^2 + \frac{f'''(x_i)}{3!}(x_{i+1} - x_i)^3 + \dots \\ &\quad \frac{f^{(n)}(x_i)}{n!}(x_{i+1} - x_i)^n + R_n \end{aligned} \tag{5.2.6}$$

and

$$R_n = \frac{f^{(n+1)}(\xi)}{(n+1)!} (x_{i+1} - x_i)^{n+1} \quad (5.2.7)$$

where x_i and x_{i+1} are two different points. If we denote $h = x_{i+1} - x_i$, then we can write the infinite series as

$$f(x_{i+1}) = f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + \frac{f'''(x_i)}{3!}h^3 + \dots + \frac{f^{(n)}(x_i)}{n!}h^n + R_n \quad (5.2.8)$$

and

$$R_n = \frac{f^{(n+1)}(\xi)}{(n+1)!} h^{n+1} \quad (5.2.9)$$

where $x_i \leq \xi \leq x_{i+1}$.

Example 5.2.1 Function Approximation

Here are some examples of well-known functions using Taylor Series expansion.

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

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

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

Sometimes Taylor series approximation is not particularly efficient. Consider a fourth-degree approximation of e^x in the interval $[-1, 1]$ expanding about $x = 0$. Then

$$P_4(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24}$$

and the error estimate is given as

$$\frac{x^5}{120} \leq e^x - P_4(x) \leq \frac{e}{120} x^5 \quad 0 \leq x \leq 1$$

$$\frac{e^{-1}}{120} |x|^5 \leq e^x - P_4(x) \leq \frac{1}{120} |x|^5 \quad -1 \leq x \leq 0$$

The error increases with increasing $|x|$ and

$$\underset{-1 \leq x \leq 1}{\text{Max}} |e^x - P_4(x)| = \frac{e}{120}$$

Summary

This chapter is an introduction to numerical analysis – the business of finding approximate solutions via a numerical technique that is implemented as a computer program. As we saw in the first four chapters, C++ provides the tools for implementing numerical techniques as robust, fast and accurate computer programs. It is important to note that usually numerical solutions are approximate for a number of reasons. In this chapter we looked at two such sources of error – truncation and round-off errors. In the later chapters we will see how these errors affect different numerical techniques.

EXERCISES

Most of the problems below involve the development of one or more functions. For each applicable case, the function prototype is given. In each case develop (a) plan to test the function(s), and (b) implement the plan in a **main** program. The functions should not use **cin** or **cout** unless specified. Put the main program in a separate file and the function(s) in separate files.

Appetizers

Problem 5.1

- (a) Compute the binary form of the following decimal numbers. (i) -187 (ii) 3009 (iii) -199 (iv) 5789.
 (b) Compute the decimal equivalent for the following binary numbers. (i) -10011 (ii) 101010 (iii) 11111100001.

Problem 5.2

Fibonacci numbers, F_i , are defined as $F_0 = F_1 = 1$ and $F_{i+2} = F_{i+1} + F_i$, $i = 1, 2, 3, \dots$. The ratio $x_n = \frac{F_{n+1}}{F_n}$ is the Golden Ratio with $n \rightarrow \infty$. It is known that $\lim_{n \rightarrow \infty} x_n = \frac{1 + \sqrt{5}}{2}$. Determine the relative error in approximating x_∞ for $n = 1, 5, 10$.

Function prototype

```
double REGoldenRatio (int n);
```

Problem 5.3

Expand the function $f(x) = 2x^4 - 1.5x^2 + 33.4x - 10.5$ in Taylor's series about $x = 0.5$. Use the resulting expression to estimate the value of $f(x=1)$ by retaining 1, 2, 3 and 4 terms of the expansion. Determine the absolute and the relative errors for each case.

Function prototype

```
void TSExpansion (int nTerms, double& dEst, double& dAbsError,
                  double& dRelError);
```

*Problem 5.4***TBC**

Main Course*Problem 5.5***TBC***Problem 5.6***TBC***Problem 5.7*

Consider the problem of estimating the area of a circle of radius R . One approach is to split the circle into a collection of uniform triangles as shown in the figure below.

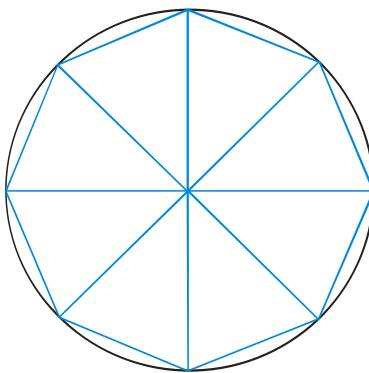


Fig. P5.7(a)

For a typical triangle that subtends an angle θ at the center of the circle as shown below

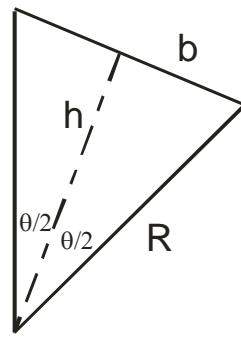


Fig. P5.7(b)

Since $b = R \sin(\theta/2)$, $h = R \cos(\theta/2)$ and $\theta = \frac{2\pi}{n}$, we have, the area of one triangle and the estimate of the area of the circle are given by

$$a_e = \frac{R^2}{2} \sin\left(\frac{2\pi}{n}\right)$$

$$A_A^{(n)} = \sum_{e=1}^n a_e = \frac{nR^2}{2} \sin\left(\frac{2\pi}{n}\right)$$

Function prototype

```
double AreaTriangleA (int n);
```

Problem 5.8

Another approach to solving the problem discussed in Problem 5.7 is shown in Fig. P5.8. Derive the expression for the estimate of the area $A_B^{(n)}$.

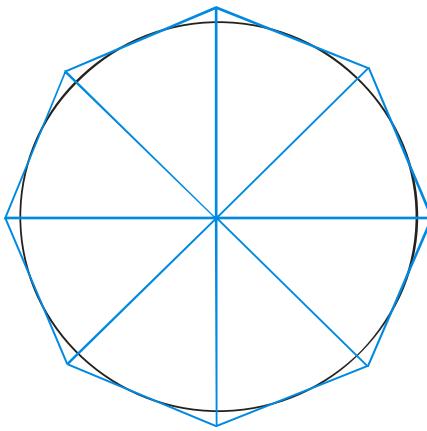


Fig. P5.8

Function prototype

```
double AreaTriangleB (int n);
```

Numerical Analysis Concepts

Problem 5.9

Most computer systems represent negative numbers as 2's complement. Find out what 2's complement is and how arithmetic operations are carried out using 2's complement.

Problem 5.10

Consider the area estimate problem discussed in Problems 5.7 and 5.8. Develop a procedure by which you can estimate the area of a circle using either Mesh A, or Mesh B or both. The input to the procedure is (a) the radius of the circle and (b) the desired accuracy. Assume that you do NOT know

the exact area of the circle. The procedure must compute the estimate for the area (within the prescribed accuracy) using the least computational effort. The computational effort, E , is defined as

$$E = 100 + \sum_{i=1}^q n_i^2$$

where q is the number of times the procedure uses the formula from either Mesh A or Mesh B, and n_i is the number of triangles in the mesh. For example, if your procedure uses Mesh A twice with $n = 10$ and $n = 20$, and Mesh B thrice with $n = 10$, $n = 25$ and $n = 50$, the total effort would be 3825.

The output from the procedure is the estimate of the area and the computational effort.

Function prototype

```
double TriangleAreaEstimate (double dR, double dError,
                           double& dComputeEffort);
```

Chapter 6

Root Finding, Differentiation and Integration

“Learning without thought is useless, thought without learning is dangerous.” Confucius .

We will look at three numerical problems commonly encountered by engineers and scientists. We will examine the various techniques to compute the roots of nonlinear functions, and learn how to carry out numerical differentiation and integration.

Objectives

- To understand how to find the roots of nonlinear functions.
- To understand the concepts associated with numerical differentiation.
- To understand the concepts associated with numerical integration.

6.1 Roots of Equations

The root of an equation is the set of values of the parameters for which the equation is satisfied. Many different types of problems can be represented as root finding problems. For instance, determining the numerical value of \sqrt{n} is equivalent to finding the positive root of the equation $x^2 - n = 0$. Drawing a circle of radius r is equivalent to plotting the roots of the equation $x^2 + y^2 - r^2 = 0$. Determining when a projectile undergoing uniform acceleration will hit the ground is equivalent to finding the (larger) root of the equation $\frac{a}{2}t^2 + v_0t + h_0 = 0$, where a is the acceleration, v_0 is the initial velocity, and h_0 is the initial height. Mathematicians have devised several methods, each with their own advantages and disadvantages, to compute the roots of equations. Here we will focus on equations of the form $f(x) = 0$. Roots of this equation are called zeros of the function.

Bracketing, Bisection Method, and False Position Method

The bisection and false position methods rely on a general concept called bracketing. If the function whose zero is to be found is continuous on an interval $[a, b]$, then according to the Intermediate Value Theorem, the zero exists in $[a, b]$ if $f(a)f(b) < 0$. The latter condition means that the function takes on values of opposite signs at a and b . Conceptually, bracketing works as follows, provided that at least one root exists in $[a, b]$.

Step 1: Pick a value in $[a, b]$ (called c) and construct two intervals, $[a, c]$ and $[c, b]$. At least one root exists in one of these intervals.

Step 2: To determine which interval, use the Intermediate Value Theorem. A root exists in $[a, c]$ if $f(a)f(c) < 0$, and in $[c, b]$ if $f(c)f(b) < 0$. One and only one of these conditions will be satisfied, because $f(a)f(b) < 0$.

Step 3: We have reduced the problem of finding a root in an interval $[a, b]$ to the (smaller) problem of finding a root in a smaller interval. Therefore, depending on the result of 2, we can set c to a or b and iterate. Essentially, we ‘bracket’ the root in smaller and smaller intervals – hence the name of the method.

Step 4: Continue 1-3 until the desired precision is reached. Clearly, if the final interval is $[a, b]$, then the root obtained by bracketing cannot differ from the actual root by more than $b - a$ in magnitude. The convergence criteria can be either the size of the interval $[a, b]$ or the magnitude of the function at c .

The bisection method and false position methods both use the pseudo-algorithm of Steps 1-4, but differ in their choice of c . The bisection method picks c to be the midpoint of $[a, b]$ - therefore, we

halve the search interval at every iteration. Consequently, of all methods using Steps 1-4, the bisection method's worst case behavior is best.

The false position method uses the more promising idea of constructing a linear approximation to the function, and picking c to be the root of the line. The line connecting $(a, f(a))$ and $(b, f(b))$ is

$$y = f(a) + \frac{f(b) - f(a)}{b - a}(x - a) \quad (6.1.1)$$

Setting y to zero and solving yields

$$c = a - f(a) \frac{b - a}{f(b) - f(a)} \quad (6.1.2)$$

Example 6.1.1

We will find the roots of the quadratic equation $x^2 - 3 = 0$. We can use $[1, 2]$ as the interval, as $f(x=1)$ is negative and $f(x=2)$ is positive. The true root lying in the given interval is 1.73205081.

Iteration	Bisection		False Position	
	a	b	a	b
1	1	2	1	2
2	1.5	2	1.66666666	2
3	1.5	1.75	1.66666666	1.83333333
4	1.625	1.75	1.66666666	1.75
5	1.6875	1.75	1.70833333	1.75
6	1.71875	1.75	1.72916666	1.75
7	1.71875	1.734375	1.72916666	1.73958333
8	1.7265625	1.734375	1.72916666	1.734375
9	1.73046875	1.734375	1.73177083	1.734375
10	1.73046875	1.73242187	1.73177083	1.73307291

20	1.73204994	1.73205184	1.73205057	1.73205184
----	------------	------------	------------	------------

We terminate the search process when the size of the interval $[a, b]$ becomes less than a specified tolerance of 10^{-5} .

Bracketing is a useful root finding method because it is guaranteed to find a root, and is easy to implement. However, due to its slow speed of convergence, mathematicians have devised faster methods. The drawback of these methods is that convergence to a root may not be guaranteed.

Approximation Methods, Secant Method, and Newton-Raphson Method

The Newton-Raphson method and the Secant Method determine roots by constructing linear approximations of the function in question. Provided that the approximation is good, the root of the linear approximation should be close to the true root. The pseudo-algorithm for all approximation methods is provided below.

Step 1: Pick m points, $(x_1, f(x_1)) \dots (x_m, f(x_m))$, where $x_1 \dots x_m$ are reasonably close to the true root.

Step 2: Construct an approximation to the function in the neighborhood of the root using the m points. Determine the root of this approximation, and call this root x_{m+1} .

Step 3: If the approximation was good, x_{m+1} should be closer to the root than any of $x_1 \dots x_m$. Therefore, we can discard x_1 , and repeat 2 with $x_2 \dots x_{m+1}$.

Step 4: Continue iterating until the desired precision is reached. A typical convergence criterion is to use the magnitude of the function at the current point and compare against the desired precision. It is helpful to use a recursive formula, if one exists, for implementation. The formula should give the root of the approximation in terms of the points used to construct the approximation: that is, $x_n = g(x_{n-m}, \dots, x_{n-1})$.

The Secant Method and Newton-Raphson method each use Steps 1-4, with different choices of m , and different approximations. Consequently, the recursive formula also differs.

The Secant Method uses the secant-line approximation to the function. Hence, $m = 2$, and the secant-line is

$$y = f(x_1) + \frac{f(x_2) - f(x_1)}{x_2 - x_1}(x - x_1) \quad (6.1.3)$$

The root of this approximation is $x_1 - f(x_1) \frac{x_2 - x_1}{f(x_2) - f(x_1)}$, and therefore the recursive formula is given by

$$x_n = g(x_{n-2}, x_{n-1}) = x_{n-2} - f(x_{n-2}) \frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} \quad (6.1.4)$$

The Newton-Raphson method, on the other hand, uses calculus to construct a tangent-line approximation to the function. Hence, $m = 1$. The tangent-line is

$$y = f(x_1) + f'(x_1)(x - x_1) \quad (6.1.5)$$

The root of the tangent-line is $x_1 - \frac{f(x_1)}{f'(x_1)}$, so the recursive formula is

$$x_n = g(x_{n-1}) = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})} \quad (6.1.6)$$

Note that the Secant Method can be considered as a special case of the Newton-Raphson Method, where the derivative $f'(x_{n-1})$ is replaced by the approximation $\frac{f(x_2) - f(x_1)}{x_2 - x_1}$.

The advantage of the approximation methods is that if they converge, they generally converge with a greater speed than the bracketing methods. This is because as $x_{n-m} \dots x_{n-1}$ approach the true root, the approximation matches the function more and more closely in the neighborhood of the true root. In fact, both the Secant Method and the Newton-Raphson Method have better than linear convergence in most cases.

The drawback of the approximation methods is that, unlike the bracketing methods, they do not guarantee convergence because the root is not confined to an interval. Hence a poor initial guess or a badly behaved function may cause the methods to fail to converge to a root. Additionally, convergence is poorer if the root is repeated (i.e. if the function can be written $f(x) = (x - \alpha)^p g(x)$, $p > 1$, where α is the root in question). The order of convergence of the Newton-Raphson Method is quadratic (i.e. 2), whereas the order of convergence of the Secant Method is the golden ratio $\frac{1 + \sqrt{5}}{2} \approx 1.618$. Calculation of the derivative for the Newton-Raphson Method, either analytically or numerically, requires extra computation.

Example 6.1.2

We will find the roots of the quadratic equation $x^2 - 3 = 0$ using the Newton-Raphson Method. Note that $f(x) = x^2 - 3$ and $f'(x) = 2x$. We will start with the initial guess for the root as $x_0 = 1$.

n	x_n	$f'(x_n)$	$f(x_n)$	$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$
0	1	2	-2	2
1	2	4	1	1.75
2	1.75	3.5	0.0625	1.73214
3	1.73214	3.46428	0.00030898	1.73205
4	1.73205	3.4641	-2.7975(10 ⁻⁶)	1.73205

We terminate the iterations when the magnitude of the function is close to zero.

It should be noted that the process can fail if at any time the value of the derivative of the function is very close to zero.

Van Wijngaarden-Dekker-Brent Method

TBC

6.2 Numerical Differentiation

As you may have seen from a calculus course, the derivative, or the rate of change of one variable with respect to another provides an understanding of the relationship between the problem variables. Numerical differentiation provides estimates for first-order and higher-order derivatives when explicit differentiation is not readily available to generate the derivative expressions.

Taylor Series Approach

Taylor's Theorem provides a means of approximating a non-polynomial function by a polynomial function. If the former function is differentiable of order $n + 1$ on an interval $[a, b]$, then it can be written as a sum of a known polynomial of degree n and indeterminate error term – a polynomial of degree $n+1$. That is,

$$f(x) = P_n(x) + R_{n+1}(x) \quad (6.2.1a)$$

$$\begin{aligned} P_n(x) &= f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2!} f''(x_0)(x - x_0)^2 + \\ &\dots + \frac{1}{n!} f^{(n)}(x_0)(x - x_0)^n \end{aligned} \quad (6.2.1b)$$

$$R_{n+1}(x) = \frac{(x - x_0)^{n+1}}{(n+1)!} f^{(n+1)}(\xi) \quad (6.2.1c)$$

where $x, x_0 \in [a, b]$, and where $\xi \in [x_0, x]$. The function $P_n(x)$ approximates $f(x)$ more and more accurately in the neighborhood of x_0 as n increases, assuming higher order derivatives are negligible compared to $(n+1)!$. This is evident when considering the remainder term; if $\frac{f^{(n+1)}(\xi)}{(n+1)!}$ is bounded by a small number, and if x is close to x_0 , then the error in the approximation is bounded by a small number as well. Consequently, we can use a Taylor polynomial to approximate a function in the neighborhood of a point for which the values of derivatives are known and for which the term $\frac{f^{(n+1)}(\xi)}{(n+1)!}$ can be ignored. The connection of Taylor's polynomial to numerical differentiation is that if higher order derivatives are negligible compared to $(n+1)!$, we can approximate derivatives of any order using Taylor series. For instance, expanding $f(x_0 + h)$ and $f(x_0 - h)$ in terms of a Taylor's polynomial of degree 2, we obtain

$$\begin{aligned} f(x_0 + h) &= f(x_0) + f'(x_0)h + f''(x_0)\frac{h^2}{2} + f'''(\xi)\frac{h^3}{6}, \xi \in [x_0, x_0 + h] \\ f(x_0 - h) &= f(x_0) - f'(x_0)h + f''(x_0)\frac{h^2}{2} - f'''(\psi)\frac{h^3}{6}, \psi \in [x_0 - h, x_0] \end{aligned} \quad (6.2.2)$$

Subtracting one equation from the other, we get

$$\begin{aligned} f'(x_0) &= \frac{f(x_0 + h) - f(x_0 - h)}{2h} + f''(\xi)\frac{h^2}{12} + \\ &\quad f'''(\psi)\frac{h^2}{12}, \xi \in [x_0, x_0 + h], \psi \in [x_0 - h, x_0] \end{aligned} \quad (6.2.3)$$

This is known as the central difference formula, because it uses two points, $f(x_0 - h)$ and $f(x_0 + h)$ that are centered about x_0 to calculate the derivative there. We can also derive forward difference and backward difference formulas by expanding $f(x_0 + h)$ and $f(x_0 - h)$ in terms of a Taylor's polynomial of degree 1. We have

$$\begin{aligned} f'(x_0) &= \frac{f(x_0 + h) - f(x_0)}{h} + f''(\xi)\frac{h}{2}, \xi \in [x_0, x_0 + h] \\ f'(x_0) &= \frac{f(x_0) - f(x_0 - h)}{h} + f''(\psi)\frac{h}{2}, \psi \in [x_0 - h, x_0] \end{aligned} \quad (6.2.4)$$

These formulas can be used to approximate the numerical derivative by assuming that second-order derivatives are negligible, for the forward and backward difference formulas, and that third-order derivatives are negligible, for the central difference formula.

Similarly, using the third degree Taylor's polynomials

$$\begin{aligned} f(x_0 + h) &= f(x_0) + f'(x_0)h + f''(x_0)\frac{h^2}{2} + f'''(x_0)\frac{h^3}{6} \\ &\quad + f^{(4)}(\xi)\frac{h^4}{24}, \xi \in [x_0, x_0 + h] \\ f(x_0 - h) &= f(x_0) - f'(x_0)h + f''(x_0)\frac{h^2}{2} - f'''(x_0)\frac{h^3}{6} \\ &\quad + f^{(4)}(\psi)\frac{h^4}{24}, \psi \in [x_0 - h, x_0] \end{aligned} \tag{6.2.5}$$

and adding these equations gives

$$\begin{aligned} f''(x_0) &= \frac{f(x_0 + h) - 2f(x_0) + f(x_0 - h)}{h^2} - f^{(4)}(\xi)\frac{h^2}{24} \\ &\quad - f^{(4)}(\psi)\frac{h^2}{24}, \xi \in [x_0, x_0 + h], \psi \in [x_0 - h, x_0] \end{aligned} \tag{6.2.6}$$

which is the central difference formula for the second derivative. Using the second degree Taylor's polynomials

$$\begin{aligned} f(x_0 + h) &= f(x_0) + f'(x_0)h + f''(x_0)\frac{h^2}{2} \\ &\quad + f'''(\xi)\frac{h^3}{6}, \xi \in [x_0, x_0 + h] \\ f(x_0 + 2h) &= f(x_0) + f'(x_0)2h + f''(x_0)\frac{(2h)^2}{2} \\ &\quad + f'''(\xi)\frac{(2h)^3}{6}, \psi \in [x_0, x_0 + 2h] \end{aligned} \tag{6.2.7}$$

and subtracting twice the first formula from the second, we get:

$$\begin{aligned} f''(x_0) &= \frac{f(x_0 + 2h) - 2f(x_0 + h) + f(x_0)}{h^2} + f'''(\xi)\frac{h}{3} \\ &\quad - f'''(\psi)\frac{4h}{3}, \xi \in [x_0, x_0 + h], \psi \in [x_0, x_0 + 2h] \end{aligned} \tag{6.2.8}$$

The general tactic to find the forward-finite-difference approximation for $f^{(n)}(x_0)$ is to manipulate the Taylor's polynomials of $f(x_0 + h) \dots f(x_0 + nh)$, so that the terms $f'(x_0) \dots f^{(n-1)}(x_0)$ cancel out. Similarly, manipulation of Taylor's polynomials for $f(x_0 - h) \dots f(x_0 - nh)$ and the cancellation of $f'(x_0) \dots f^{(n-1)}(x_0)$ will result in the backward-finite-difference approximation formulas. Finally, for the central-finite-difference approximation for $f^{(n)}(x_0)$, $f(x_0 - mh) \dots f(x_0 + mh)$ should be manipulated so that lower-order derivatives disappear, where $m = \frac{n}{2}$ for n even, and $m = \frac{n+1}{2}$ for n odd. Clearly, these manipulations become increasingly more tedious as n increases.

Difference Formulas

Using difference operators provides a much easier approach to constructing the same formulas as above. The drawback is that the formulas are approximations, and no explicit error term is present. The main idea is to use difference operators to approximate the derivative operator, $\frac{d}{dx}$. The three approximations are:

$$\frac{d}{dx} \approx \frac{\Delta}{\Delta x} (\text{forward}), \frac{d}{dx} \approx \frac{\delta}{\delta x} (\text{central}), \frac{d}{dx} \approx \frac{\nabla}{\nabla x} (\text{backward}) \quad (6.2.9)$$

where the operators themselves mean:

$$\begin{aligned} \Delta f(x_i) &= f(x_{i+1}) - f(x_i) \\ \delta f(x_i) &= f(x_{i+1/2}) - f(x_{i-1/2}) \\ \nabla f(x_i) &= f(x_i) - f(x_{i-1}) \end{aligned} \quad (6.2.10)$$

We will assume that the distance between successive x values is constant, so $\Delta x_i = \nabla x_i = \delta x_i = h$. Using these operators, we can easily obtain the approximations to the first derivative

$$\text{Forward : } \frac{df}{dx} \approx \frac{\Delta f}{\Delta x} = \frac{f(x_{i+1}) - f(x_i)}{\Delta x} = \frac{f(x_i + h) - f(x_i)}{h} \quad (6.2.11)$$

$$\text{Central : } \frac{df}{dx} \approx \frac{\delta f}{\delta x} = \frac{f(x_{i+1/2}) - f(x_{i-1/2})}{\delta x} = \frac{f(x_i + h) - f(x_i - h)}{2h} \quad (6.2.12)$$

$$\text{Backward : } \frac{df}{dx} \approx \frac{\nabla f}{\nabla x} = \frac{f(x_i) - f(x_{i-1})}{\nabla x} = \frac{f(x_i) - f(x_i - h)}{h} \quad (6.2.13)$$

The reason that the step size is doubled for the central-finite-difference approximation is that the values of the function at $f(x_i + h/2)$ and $f(x_i - h/2)$ may not be known.

Similarly, for the second derivative approximations,

$$\text{Forward : } \frac{d^2f}{dx^2} \approx \frac{\Delta}{\Delta x} \left[\frac{\Delta f}{\Delta x} \right] = \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{(\Delta x)^2} = \frac{f(x_i + 2h) - 2f(x_i + h) + f(x_i)}{h^2} \quad (6.2.14)$$

$$\text{Central : } \frac{d^2f}{dx^2} \approx \frac{\delta}{\delta x} \left[\frac{\delta f}{\delta x} \right] = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{(\delta x)^2} = \frac{f(x_i + h) - 2f(x_i) + f(x_i - h)}{h^2} \quad (6.2.15)$$

$$\text{Backward : } \frac{d^2f}{dx^2} \approx \frac{\nabla}{\nabla x} \left[\frac{\nabla f}{\nabla x} \right] = \frac{f(x_i) - 2f(x_{i-1}) + f(x_{i-2})}{(\nabla x)^2} = \frac{f(x_i) - 2f(x_i - h) + f(x_i - 2h)}{h^2} \quad (6.2.16)$$

In general, the error for the forward difference and backward difference formulae is of the order $O(h)$. The error for the central difference formula is of the order $O(h^2)$. Consequently, the central difference formula is more accurate than the other two techniques as we will see in following example. It should be noted that the accuracy can be increased by using additional sampling points around the point of interest.

Example 6.2.1 Forward Difference

Compute the derivative of function $f(x) = x^3 - 2x^2 + 10x - 5$ at $x = 2$.

Note that the analytical derivative is $f'(x) = 3x^2 - 4x + 10$ and $f'(x=2) = 14$. The table below shows the calculations using Eqn. (6.2.11). The relative error is defined as $\frac{f_{FD} - f_{exact}}{f_{exact}}$ where $f_{exact} = 14$.

h	f(x+h)	f(x)	f'(x)	Rel Error
1.000000E-15	1.500000000000000E+01	1.500000000000000E+01	1.065814E+01	-2.387042E-01
1.000000E-10	1.500000001400E+01	1.500000000000000E+01	1.400000E+01	8.274037E-08
1.000000E-08	1.500000140000E+01	1.500000000000000E+01	1.400000E+01	6.610792E-09
1.000000E-05	1.5000140000400E+01	1.500000000000000E+01	1.400004E+01	2.857156E-06

1.000000E-02	1.5140401000000E+01	1.50000000000000E+01	1.404010E+01	2.864286E-03
1.000000E-01	1.6441000000000E+01	1.50000000000000E+01	1.441000E+01	2.928571E-02

Example 6.2.2 Backward Difference

Redo Example 6.2.1. The table below shows the calculations using Eqn. (6.2.13). The relative error is defined as $\frac{f_{BD} - f_{exact}}{f_{exact}}$ where $f_{exact} = 14$.

h	f(x-h)	f(x)	f'(x)	Rel Error
1.000000E-15	1.50000000000000E+01	1.50000000000000E+01	1.421085E+01	1.506105E-02
1.000000E-10	1.4999999998600E+01	1.50000000000000E+01	1.400000E+01	8.274037E-08
1.000000E-08	1.4999999860000E+01	1.50000000000000E+01	1.400000E+01	6.610792E-09
1.000000E-05	1.4999860000400E+01	1.50000000000000E+01	1.399996E+01	-2.857130E-06
1.000000E-02	1.4860399000000E+01	1.50000000000000E+01	1.396010E+01	-2.850000E-03
1.000000E-01	1.3639000000000E+01	1.50000000000000E+01	1.361000E+01	-2.785714E-02

Example 6.2.3 Central Difference

Redo Example 6.2.1. The table below shows the calculations using Eqn. (6.2.12). The relative error is defined as $\frac{f_{CD} - f_{exact}}{f_{exact}}$ where $f_{exact} = 14$.

h	f(x-h)	f(x+h)	f'(x)	Rel Error
1.000000E-15	1.50000000000000E+01	1.50000000000000E+01	1.243450E+01	-1.118216E-01
1.000000E-10	1.4999999998600E+01	1.5000000001400E+01	1.400000E+01	8.274037E-08
1.000000E-08	1.4999999860000E+01	1.5000000140000E+01	1.400000E+01	6.610792E-09
1.000000E-05	1.4999860000400E+01	1.5000140000400E+01	1.400000E+01	1.289521E-11
1.000000E-02	1.4860399000000E+01	1.5140401000000E+01	1.400010E+01	7.142857E-06
1.000000E-01	1.3639000000000E+01	1.6441000000000E+01	1.401000E+01	7.142857E-04

The following table shows the difference in results using the three techniques. The central difference technique is by far the most accurate with the smallest error.

h	FD	BD	CD	Best
1.000000E-15	-2.39E-01	1.51E-02	-1.12E-01	BD
1.000000E-10	8.27E-08	8.27E-08	8.27E-08	All
1.000000E-08	6.61E-09	6.61E-09	6.61E-09	All
1.000000E-05	2.86E-06	-2.86E-06	1.29E-11	CD
1.000000E-02	2.86E-03	-2.85E-03	7.14E-06	CD
1.000000E-01	2.93E-02	-2.79E-02	7.14E-04	CD

6.3 Numerical Integration

Consider the problem of evaluating the integral

$$I = \int_a^b F(x) dx \quad (6.3.1)$$

We will assume that we either know the function $F(x)$ that is difficult to integrate exactly or that we have a set of discrete data (points where the function has been evaluated numerically). In either case, we need to develop a numerical technique to evaluate the integral.

The basic idea in numerical integration is to construct another function $P(x)$ (usually a polynomial) that is a suitable approximation of $F(x)$ and is simple to integrate. The interpolating polynomial of degree n , denoted P_n , is such that it interpolates the integrand at $(n+1)$ points in the interval $[a, b]$. While there exist errors, $E = F(x) - P_n(x)$, the error may not always be of the same sign so that the overall error is small.

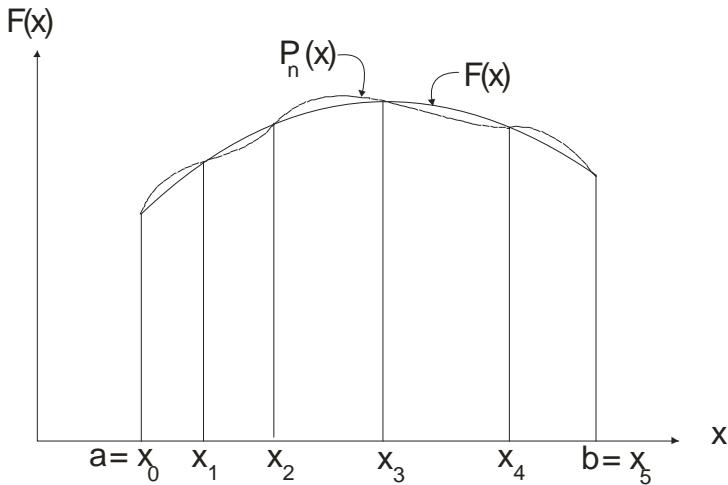


Fig. 6.3.1 Original function and its polynomial approximation

Hence the equivalent integral is given by

$$I = \int_a^b F(x) dx \approx \int_a^b P_n(x) dx \quad (6.3.2)$$

Fig. 6.3.1 shows the situation where the discrete values are known at 6 points and the approximate function $P_n(x)$ is made to pass through those six points. In more general terms, we could have a number of scenarios.

- (1) We know the data at exactly $(n+1)$ points and we fit a polynomial of degree n that passes through those points (as shown in Fig. 6.3.1).
- (2) We know the data at more than $(n+1)$ points and we fit a polynomial of degree n using a concept such as least-squares fit. We will look at least-squares fit in Chapter 11.

- (3) If, on the other hand, we know the function $F(x)$, then we can evaluate the function at $(n+1)$ points and use the approach associated with scenario (1).

Newton-Cotes

When the function to be integrated is known at equally spaced points, we can use the forward difference polynomial (see Section 6.2) and fit the data. Recall that

$$\begin{aligned} P_n(x_0 + sh) &= f_0 + s\Delta f_0 + \frac{s(s-1)}{2}\Delta^2 f_0 + \dots \\ &\quad + \frac{s(s-1)(s-2)\dots[s-(n-1)]}{n!}\Delta^n f_0 + \text{error} \end{aligned} \quad (6.3.3)$$

where

$$x = x_0 + sh \quad (6.3.4a)$$

$$\text{error} = \binom{s}{n+1} h^{n+1} f^{(n+1)}(\xi) \quad x_0 \leq x \leq x_n \quad (6.3.4b)$$

Hence,

$$I = \int_a^b F(x) dx \approx \int_a^b P_n(x) dx = h \int_{s(a)}^{s(b)} P_n(s) ds \quad (6.3.5)$$

We can match the limits of integration by recognizing that the point $x=a$ corresponds to $s=0$ and $x=b$ corresponds to $s=s$. Hence Eqn. (6.3.5) we have

$$I = h \int_0^s P_n(x_0 + sh) ds \quad (6.3.6)$$

The value of n determines the different Newton-Cotes scheme and hence the obtained precision.

Trapezoidal Rule

We obtain this rule by fitting a linear polynomial to two discrete points. From Fig. 6.3.2, we need to compute the shaded area. The upper limit of integration x_1 corresponds to $s=1$. We can rewrite Eqn. (6.3.6) as

$$I_1 = h \int_{x_0}^{x_1} (f_0 + s\Delta f_0) ds = h \left[sf_0 + \frac{s^2}{2} \Delta f_0 \right]_{x_0}^{x_1} \quad (6.3.7)$$

where the left hand represents the integral only for the first interval. Denoting $\Delta f_0 = f_1 - f_0$, we can rewrite the above equation as

$$I_1 = \frac{1}{2}h(f_0 + f_1) \quad (6.3.8)$$

and represent the entire integral as

$$I = \sum_{i=1}^n I_i = \sum_{i=1}^n \frac{1}{2}h_i(f_{i-1} + f_i) \quad (6.3.9)$$

where

$$h_i = x_i - x_{i-1} \quad (6.3.10)$$

We can further simplify the formula if we assume that the points are equally spaced.

$$I = \frac{1}{2}h(f_0 + 2f_1 + 2f_2 + \dots + 2f_{n-1} + f_n) \quad (6.3.11)$$

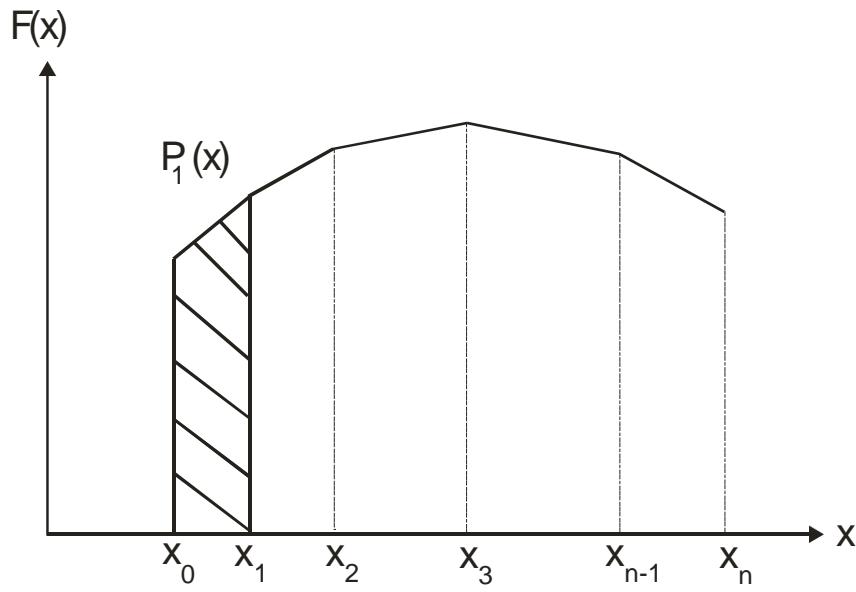


Fig. 6.3.2 Trapezoidal Rule

We can compute the error by integrating Eqn. (6.3.4b) as follows.

$$\text{Error} = h \int_0^{s-1} \frac{s(s-1)}{2} h^2 f''(\xi) ds = -\frac{1}{12} h^3 f''(\xi) = O(h^3) \quad (6.3.12)$$

The total error for equally spaced data is given by

$$\sum_{i=1}^n \text{Error} = \sum_{i=1}^n -\frac{1}{12} h^3 f''(\xi) = n \left(-\frac{1}{12} h^3 f''(\bar{\xi}) \right) \quad (6.3.13)$$

where $x_0 \leq \bar{\xi} \leq x_n$. The number of increments $n = \frac{x_n - x_0}{h}$. Therefore

$$\text{Total Error} = -\frac{1}{12} (x_n - x_0) h^2 f''(\bar{\xi}) = O(h^2) \quad (6.3.14)$$

Example 6.3.1 Trapezoidal Rule

Evaluate $\int_{-1}^1 x^4 dx$. The exact answer is 0.4. We will compute the integral for different values of sampling points, $n+1$. Note that $h = \frac{b-a}{n}$.

$$(a) n=1, h = \frac{1-(-1)}{1} = 2. I = \frac{1}{2}(2)(f_0 + f_1) = \left[(-1)^4 + (1)^4 \right] = 2$$

$$(b) n=2, h = \frac{1-(-1)}{2} = 1. I = \frac{1}{2}(1)(f_0 + 2f_1 + f_2) = \frac{1}{2} \left[(-1)^4 + 2(0)^4 + (1)^4 \right] = 1$$

$$(c) n=4, h = \frac{1-(-1)}{4} = 0.5.$$

$$\begin{aligned} I &= \frac{1}{2}(0.5)(f_0 + 2f_1 + 2f_2 + 2f_3 + f_4) \\ &= \frac{1}{4} \left[(-1)^4 + 2(-0.5)^4 + 2(0)^4 + 2(0.5)^4 + (1)^4 \right] = 0.5625 \end{aligned}$$

Simpson's Rule

We obtain this rule by fitting a quadratic polynomial through three equally spaced points. Fig. 6.3.3 shows the shaded area arising from this computation. We can write the integral as

$$I_1 = h \int_{x_0}^{x_2} \left(f_0 + s\Delta f_0 + \frac{s(s-1)}{2} \Delta^2 f_0 \right) ds = \frac{1}{3} h (f_0 + 4f_1 + f_2) \quad (6.3.12)$$

where the left hand represents the integral only for the first interval. As before we can represent the entire integral as

$$I = \frac{1}{3} h (f_0 + 4f_1 + 2f_2 + 4f_3 + \dots + 2f_{n-2} + 4f_{n-1} + f_n) \quad (6.3.13)$$

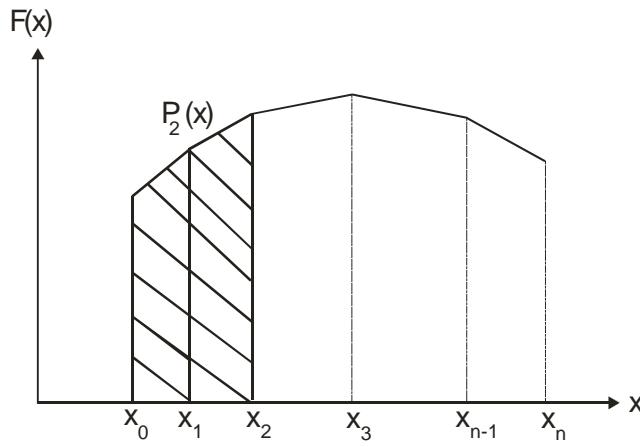


Fig. 6.3.3 Simpson's Rule

Example Program 6.3.2 Trapezoidal and Simpson's Rule

Evaluate $\int_{-1}^1 x^4 dx$. We will write a computer program to evaluate the integral for different values of sampling points, n , $1 \leq n \leq 1024$.

main.cpp

```

1  #include <iostream>
2  #include <iomanip>
3  #include <cmath>
4
5  double Func (double x) //function to integrate
6  {
7      return pow(x,4);
8  }
9
10 int main ()
11 {
12     const int nLow=1, nHigh=1024;           // controls n
13     const double dLow = -1.0, dHigh = 1.0; // integ. limits
14     double dTrapRule, dSimpsonRule, dFactor;
15
16     // output table heading
17     std::cout << "      n      " << "      "
18     << "    Trapezoid    " << "      "
19     << "    Simpson    " << '\n';
20
21     for (int n=nLow; n <= nHigh; n*=2)
22     {
23         double h = (dHigh-dLow)/static_cast<double>(n);
24         dTrapRule = dSimpsonRule = Func(dLow); // left-end
25
26         for (int i=2; i <= n; i++) // internal points
27         {
28             double dX = dLow + static_cast<double>(i-1)*h;
29             dFactor = 2.0;
30             dTrapRule += dFactor*Func(dX);
31             dFactor = (i%2 == 0? 4.0 : 2.0);
32             dSimpsonRule += dFactor*Func(dX);
33         }
34
35         dTrapRule += Func(dHigh); // right-end
36         dTrapRule *= (h/2.0);
37         dSimpsonRule += Func(dHigh); // right-end
38         dSimpsonRule *= (h/3.0);
39
40         // show the results
41         std::cout << std::setw(4) << n << "      "
42         << std::setw(15) << dTrapRule << "      "
43         << std::setw(15) << dSimpsonRule << '\n';
44     }
45     std::cout << std::endl;
46
47     return 0;
48 }
```

The output from the computer program is shown in Fig. 6.3.4.

n	Trapezoid	Simpson
1	2	1.33333
2	1	0.666667
4	0.5625	0.416667
8	0.441406	0.401042
16	0.4104	0.400065
32	0.402603	0.400004
64	0.400651	0.4
128	0.400163	0.4
256	0.400041	0.4
512	0.40001	0.4
1024	0.400003	0.4

Press any key to continue . . .

Fig. 6.3.4 Output from Example 6.3.2

As the results show, for the same computational effort, Simpson's Rule is more accurate than Trapezoidal Rule.

Gauss-Legendre Quadrature

The traditional Newton-Cotes techniques such as Trapezoidal Rule or Simpson's Rule are not as efficient or accurate as the Gauss-Legendre Quadrature.

The base points x_i and the weights w_i are chosen so that the sum of the $(n+1)$ appropriately weighted values of the function yields the integral exactly when $F(x)$ is a polynomial of degree $(2n+1)$ or less.

$$\int_a^b F(x) dx = \int_{-1}^1 \hat{F}(\xi) d\xi = \sum_{i=1}^n w_i \hat{F}(\xi_i) \quad (6.3.14)$$

where ξ_i are the base points (or, roots of the Legendre polynomial $P_{n+1}(\xi)$), and

$$F(x)dx = F(x(\xi)) \cdot \frac{dx}{d\xi} d\xi = F(x(\xi)) \cdot J(\xi) d\xi = \hat{F}(\xi) d\xi \quad (6.3.15)$$

where J is the Jacobian. The following two points should be noted

- (a) Gauss-Legendre is more efficient because it requires fewer base points to achieve the same level of accuracy as the Newton-Cotes methods, and
- (b) The error is zero if the $(2n + 2)^{th}$ derivative of the integrand vanishes. Or, a polynomial of degree n is integrated exactly by employing $(n + 1)/2$ Gauss points.

Table 6.3.1 Gauss points and weights

Order, n	Weight	Location
1	2.0	0.0
2	1.0	0.57735 02691
	1.0	-0.57735 02691
3	0.55555 55555	0.77459 66692
	0.55555 55555	-0.77459 66692
	0.88888 88888	0.0

Example 6.3.3

Evaluate $\int_{-1}^1 x^4 dx$. The exact answer is 0.4.

We have $F(x) = F(\xi) = x^4$. No Jacobian is needed.

Using $n = 1$: $w_1 = 2.0$ and $\xi_1 = 0.0$. Hence $I = (2.0)(0.0)^4 = 0.0$.

Using $n = 2$: $(w_1, \xi_1) = (1.0, 0.5773502691)$ and $(w_2, \xi_2) = (1.0, -0.5773502691)$.

Hence $I = (1.0)(0.5773502691)^4 + (1.0)(-0.5773502691)^4 = 0.222222222$.

Using $n = 3$: $(w_1, \xi_1) = (0.5555555555, 0.7745966692)$

$(w_2, \xi_2) = (0.5555555555, -0.7745966692)$

$(w_3, \xi_3) = (0.8888888888, 0.0)$

Hence $I = (0.5555555555)(0.7745966692)^4 + (0.5555555555)(-0.7745966692)^4$

$+ (0.8888888888)(0.0)^4 = 0.4$

The original function is a polynomial of degree 4 and can be integrated exactly by Gauss Quadrature rule $\frac{(n+1)}{2} = \frac{4+1}{2} = 3$ (note that the rounding takes place to the next highest integer). The results bear out the rule.

Example 6.3.4

Evaluate $\int_{-1}^1 \left[\frac{2x-1}{x^2 - 6x + 13} \right] dx$.

The integrand is not a polynomial. The exact solution is -0.1119 . We will solve with $n=2$ and $n=3$ rules as shown below.

Order, n	w_i	ξ_i	$f(\xi_i)$	$w_i f(\xi_i)$
2	1.0	0.57735 02691	0.015675	0.015675
	1.0	-0.57735 02691	-0.128276	-0.128276
			TOTAL	-0.112601
3	0.55555 55555	0.77459 66692	0.0613458	0.034081
	0.55555 55555	-0.77459 66692	-0.1397	-0.0776113
	0.88888 88888	0.0	-0.0769231	-0.0683761
			TOTAL	-0.111906

Example 6.3.5

Evaluate $\int_1^7 \frac{1}{x} dx$. Exact: $I = \int_1^7 \frac{1}{x} dx = \ln(7) = 1.94591$

In order to use G-Q Rule we must first construct a mapping function to map the given domain $[1, 7]$ to the required domain $[-1, 1]$. The mapping function is

$$x = \frac{1-\xi}{2} x_1 + \frac{1+\xi}{2} x_2 = \frac{1-\xi}{2}(1) + \frac{1+\xi}{2}(7) = 4 + 3\xi$$

$$\frac{dx}{d\xi} = 3$$

Hence,

$$\int_1^7 \frac{1}{x} dx = \int_{-1}^1 \frac{1}{4+3\xi} 3d\xi = \sum_{i=1}^n w_i f(\xi_i)$$

Using $n=4$ rule, we have the following results.

psi	w	f(psi)	w*f(psi)
-0.861136312	0.34785484	2.11776	0.736673
-0.339981044	0.65214549	1.006692	0.65651
0.339981044	0.65214549	0.597616	0.389733
0.861136312	0.34785484	0.455691	0.158514
	Sum=		1.94143

Note $I_1 = 1.5$, $I_2 = 1.846$, $I_3 = 1.9245$ and $I_5 = 1.944981413$.

Two-Dimensional Functions: Functions that involve two independent natural coordinates are handled in a manner similar to one-dimensional functions.

$$\begin{aligned} \int_c^d \int_a^b F(x, y) dx dy &= \int_{-1}^1 \int_{-1}^1 F(x(\xi, \eta), y(\xi, \eta)) |J| d\xi d\eta \\ &= \sum_{j=1}^n \sum_{i=1}^n w_i w_j f(\xi_i, \eta_j) \end{aligned} \quad (6.3.16a)$$

$$\text{where } f(\xi_i, \eta_j) = F(x(\xi, \eta), y(\xi, \eta)) |J| \quad (6.3.16b)$$

$$|J| = \det(\mathbf{J}) \quad \text{where } \mathbf{J}_{2 \times 2} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{bmatrix} \quad (6.3.16c)$$

The values of the weights and natural coordinates are the same as shown in Table 6.3.1 except that ξ in the table refers to both ξ and η .

Example 6.3.6

Evaluate $\int_{-1}^1 \int_{-1}^1 x^2 dx dy$. The exact answer is $\frac{4}{3}$.

Using the $n = 2$ rule, we have four Gauss points.

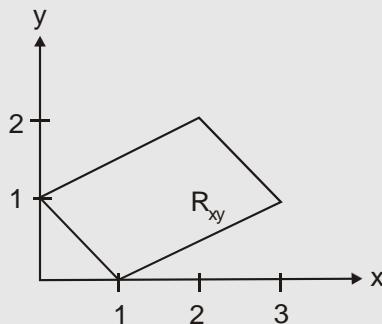
ξ_i	η_j	w_i	w_j	$f(\xi_i, \eta_j)$	$w_i w_j f(\xi_i, \eta_j)$
-0.5773502691	-0.57735 02691	1.0	1.0	1/3	1/3

0.5773502691	-0.57735 02691	1.0	1.0	1/3	1/3
0.5773502691	0.57735 02691	1.0	1.0	1/3	1/3
-0.5773502691	0.57735 02691	1.0	1.0	1/3	1/3
				TOTAL	4/3

The answer obtained is the exact answer. Once again, the appropriate quadrature order is $\frac{(n+1)}{2} = \frac{(2+1)}{2} = 2$ and using the rule leads to the exact answer.

Example 6.3.7

Evaluate $I = \iint_{R_{xy}} (x+y)^3 dxdy$



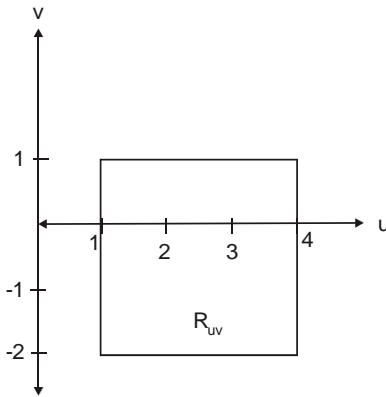
Analytical Approach: The sides of the domain are not parallel to the axes making it difficult to set the limits of integration. However, the sides are such that

$$x + y = c_1 \quad \text{and} \quad x - 2y = c_2$$

Therefore, we can introduce two new variables and set up a mapping such that

$$u = x + y \quad \text{and} \quad v = x - 2y$$

The transformed domain is shown below.



$$\text{Hence, } I = \iint_{R_{xy}} (x+y)^3 dx dy = \iint_{R_{uv}} (u)^3 \det(J) du dv$$

$$\det(J) = \frac{\partial(x, y)}{\partial(u, v)} = \frac{1}{\frac{\partial(u, v)}{\partial(x, y)}} = \frac{1}{\begin{vmatrix} 1 & 1 \\ 1 & -2 \end{vmatrix}} = \frac{1}{3}$$

$$\text{Substituting, } I = \iint_{R_{uv}} (u)^3 \det(J) du dv = \int_{-2}^1 \int_1^4 \frac{u^3}{3} du dv = \frac{765}{12} = 63.75$$

Numerical Approach: We will construct the appropriate mapping functions to map the given integration domain into a square $\xi \in [-1, 1], \eta \in [-1, 1]$. Using the following mapping functions

$$x = \sum_{i=1}^4 \phi_i(\xi, \eta) x_i \quad y = \sum_{i=1}^4 \phi_i(\xi, \eta) y_i$$

$$\text{where } \phi_1 = \frac{1}{4}(1-\xi)(1-\eta) \quad \phi_2 = \frac{1}{4}(1+\xi)(1-\eta)$$

$$\phi_3 = \frac{1}{4}(1+\xi)(1+\eta) \quad \phi_4 = \frac{1}{4}(1-\xi)(1+\eta)$$

The given domain is such that $(x_1, y_1) = (1, 0)$, $(x_2, y_2) = (3, 1)$, $(x_3, y_3) = (2, 2)$ and $(x_4, y_4) = (0, 1)$. The jacobian can be constructed as

$$\mathbf{J}_{2 \times 2} = \frac{1}{4} \begin{bmatrix} 4 & 2 \\ -2 & 2 \end{bmatrix} \quad \det(\mathbf{J}) = \frac{3}{4}$$

(a) We will use the one-point rule first.

$$i = j = 1 \quad w_i = w_j = 2.0 \quad (\xi_i, \eta_j) = (0, 0)$$

$$x + y = \left(\sum_{i=1}^4 \phi_i x_i \right) + \left(\sum_{i=1}^4 \phi_i y_i \right) = \left(\frac{1}{4}(1+3+2+0) + \frac{1}{4}(0+1+2+1) \right) = \frac{5}{2}$$

$$I = (2)(2) \left(\frac{5}{2} \right)^3 \left(\frac{3}{4} \right) = 46.875$$

(b) Now the two point rule. The details of the calculations are shown below.

ξ_i	η_j	w_i	w_j	ϕ_1	ϕ_2	ϕ_3	ϕ_4	x	y	$(x + y)^3$
-0.57735	-0.57735	1.0	1.0	0.622008	0.166667	0.044658	0.166667	1.21133	0.42265	4.362508
0.57735	-0.57735	1.0	1.0	0.166667	0.622008	0.166667	0.044658	2.36603	1	38.13748
0.57735	0.57735	1.0	1.0	0.044658	0.166667	0.622008	0.166667	1.78868	1.57735	38.13748
-0.57735	0.57735	1.0	1.0	0.166667	0.044658	0.166667	0.622008	0.63398	1	4.362508
								$I = \det(J) \sum (x + y)^3$		63.75

6.4 User-Defined Functions

In the preceding sections we saw three numerical analysis topics. Each topic has solution techniques associated with it that should be implemented in a numerical analysis library in as general a manner as possible. Clearly we would like to separate the implementation of the numerical analysis technique from the user provided or generated information. For example, Newton-Raphson is a general technique for finding a root of any function not a specific function. If we develop the source code for Newton-Raphson technique then how do we compute the function and gradient values that are needed to obtain the solution?

The solution is in the use of function pointers and callback functions. We will see pointers in Chapter 8 and understand how best to use them. However, at this stage it should not prevent us from using the concept in developing and writing effective, general-purpose source code. We will illustrate the idea using the Newton-Raphson technique and an example.

Example 6.4.1

We will illustrate the ideas for a general-purpose numerical analysis interface for obtaining values from user-defined code through the use of Newton-Raphson technique. The specific example is to compute a root of the function

$$f(x) = (x - 2.3)(x - 4.56)(x - 3.7)$$

Step 1: Construct the gateway to Newton-Raphson technique. By this we mean, we will develop the interface or function prototype.

```
bool NewtonRaphson (double& dRoot, const int nMaxIter,
                     const double dConvTol,
                     void(*userfunc)(double dx, double& dFX, double& dDX));
```

The function returns `true` if the root is found, `false` otherwise. The function arguments are as follows.

<code>dRoot</code>	Root. Input is the initial guess and the returned value is the estimate of the root.
<code>nMaxIter</code>	Maximum number of iterations.
<code>dConvTol</code>	Convergence tolerance.
<code>*userfunc</code>	Pointer to the function that will be called to compute the function value (<code>dFX</code>) and derivative value (<code>dDX</code>) at the current point (<code>dx</code>). One must pay particular attention to this function – (a) the function prototype is like any other function, and (b) it helps to have the minimum number of arguments in the function call. In this particular case, the function is passed the value of the current point and the function returns the function and the first derivative values at the current point.

Step 2: Create the function.

```
1 #include <cmath>
2 #include "newtonraphson.h"
3
4 const double TOL = 1.0e-6; // limit on derivative
5
6 bool NewtonRaphson (double& dRoot, const int nMaxIterations,
7                      const double dConvTol,
8                      void(*userfunc)(double dx, double& dFX, double& dDX))
9 {
10    int i;
11    double dFX, dDX; // function and first derivative at current point
12
13    // loop thro' all iterations
14    for (i=1; i <= nMaxIterations; i++)
15    {
16        // compute f, df at current point
17        userfunc (dRoot, dFX, dDX);
18    }
}
```

```

19         // if derivative is near zero, return with error message
20     if (fabs(dDX) < TOL)
21         return false;
22
23         // compute new point
24     double dNewRoot = dRoot - (dFX/dDX);
25
26         // convergence check
27     if (fabs(dNewRoot - dRoot) <= dConvTol || 
28         fabs(dFX) <= dConvTol)
29     {
30         dRoot = dNewRoot;
31         return true;
32     }
33
34         // update estimate of root
35     dRoot = dNewRoot;
36 }
37
38         // did not converge
39     return false;
40 }
```

The interesting line in the source code is line 17 where the user function is called to compute the function and its derivative at the current point.

Step 3: Create the program to call the function.

```

1 #include <iostream>
2 #include "newtonraphson.h"
3
4 void MyFunction (double dx, double& dFX, double& dDX)
5 {
6     // function
7     dFX = (dx-2.3)*(dx+4.56)*(dx-3.7);
8
9     // first derivative
10    dDX = (dx+4.56)*(dx-3.7) +
11        (dx-2.3)*(dx-3.7) +
12        (dx-2.3)*(dx+4.56);
13 }
14
15 int main ()
16 {
17     const int MAXITERATIONS = 100;
18     const double CONVERGENETOL = 1.0e-6;
19     double dRoot;
20
21     // get initial guess from user
22     std::cout << "Initial guess: ";
23     std::cin >> dRoot;
24
25     // get the root
26     if (NewtonRaphson (dRoot, MAXITERATIONS, CONVERGENETOL, MyFunction))
27         std::cout << "Root is : " << dRoot << '\n';
```

```
28     else
29         std::cout << "Cannot find the root\n";
30
31     return 0;
32 }
```

The user-defined function is contained in lines 4-13 and is called `MyFunction`. It is used as the fourth argument in the call to the `NewtonRaphson` function in line 26.

We will look at more advanced scenarios involving object-oriented design in Chapter 13 (Section 13.4).

EXERCISES

Most of the problems below involve the development of one or more functions. In each case develop (a) plan to test the function(s), and (b) implement the plan in a **main** program. The functions should not use **cin** or **cout** unless specified. Put the main program in a separate file and the function(s) in separate files.

Appetizers

Problem 6.1

TBC

Main Course

TBC

Numerical Analysis Concepts

TBC

Classes: Objects 101

"The person who knows "how" will always have a job. The person who knows "why" will always be his boss." Diane Ravitch

"Real learning comes about when the competitive spirit has ceased." J. Krishnamurti

Classes and objects were introduced in Chapter 1. It cannot be overemphasized that proper definition and use of classes can lead to increased productivity with all aspects of software engineering. In this chapter we will begin the long, systematic process of understanding what classes are and how to use them effectively in computer programs simple and complex. In other words, we will try to understand why object-oriented programming (OOP) is the choice for developing useful programs. More advanced concepts will be covered in Chapters 8, 9 and 13.

Objectives

- To understand what are objects and classes.
- To understand what is data abstraction.
- To learn how to define and use classes.
- To learn more about the standard `string` class.

7.1 A Detour – Why OOP?

It is unlikely that a program refers to a single object or entity. Most practical situations involve several different objects that interact with each other. Let us look at the following problem.

Problem Statement: Given the required data of a typical planar truss, we are required to develop the procedure to draw the truss on a graph paper. A truss is made up of one or more members. These members are straight and slender. Each member is connected to two joints. Each joint is located in the coordinate space such that each joint has (x, y) coordinate values. To identify these members and joints in a truss, we used simple integers starting at 1 so that the joints were numbered 1, 2, 3, ... and the elements were numbered 1, 2, 3, ... It was further stipulated that no two joints would have the same identification number and the same applies to the members.

To simplify the issue, we can look at the truss as a collection of points (joints) and straight lines (members). A graphical representation of the truss in this manner is known in computer graphics as a wireframe representation as shown in Fig. 7.1.1.

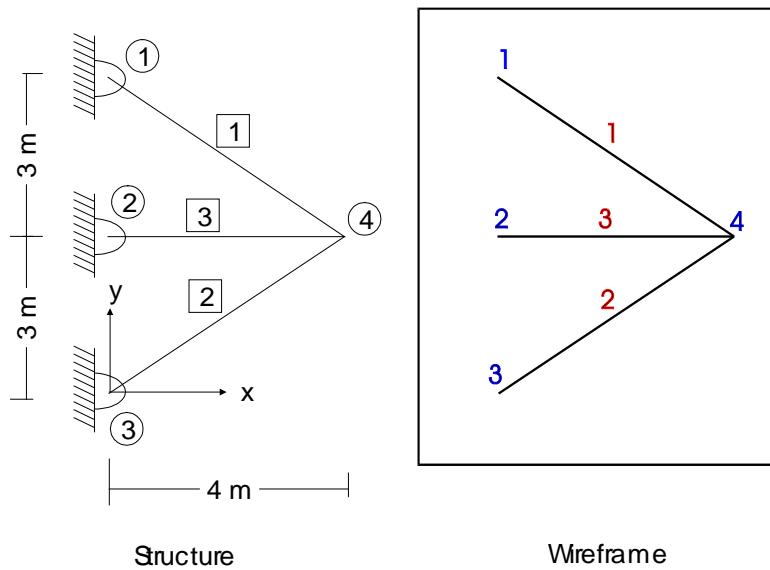


Fig. 7.1.1 Truss and its wireframe representation

Solution: We will now develop the general algorithm to meet our objectives.

Variable Dictionary

nPoints	Number of points
nLines	Number of lines

Algorithm

- (1) Obtain the total number of points, nPoints, and total number of lines, nLines.
- (2) Obtain the (x, y) coordinates of each point. Obtain the (start point, end point) numbers for each line. Track the smallest and the largest (x, y) values as (x_{\min}, x_{\max}) and (y_{\min}, y_{\max}) .
- (3) Compute the scale for the graph based on (x_{\min}, x_{\max}) and (y_{\min}, y_{\max}) values. For an anisotropic scaling, the scaling values are different in the x and the y directions. For an isotropic scaling, there is one value that is the smaller of the two scaling values.
- (4) Loop through all the lines, i .
- (5) For the current line, obtain the start point number. Compute the graph coordinates (x_{gs}, y_{gs}) . Move to (x_{gs}, y_{gs}) .
- (6) Obtain the end point number. Compute the graph coordinates (x_{ge}, y_{ge}) . Draw the line to (x_{ge}, y_{ge}) .
- (7) End loop i .

We will now tackle the problem of translating this algorithm into a computer program. An examination of the algorithm shows that there are two major pieces of information (entities) that we must handle – data associated with points, and data associated with lines. Several questions arise naturally. The most obvious one is “What data structure should be used?”, or “How should the problem data be stored?”. Let us assume that we store the point and line data in arrays (vectors to be specific) as shown below. We will name this approach *Array-based Solution*. The index of the vector will provide the data access mechanism.

Array-based Solution

Array	Size	Remarks
fVX	nPoints	Vector containing the global x-coordinates of all the points
fVY	nPoints	Vector containing the global y-coordinates of all the points
nVSP	nLines	Vector containing the start point number of all the lines
nVEP	nLines	Vector containing the end point number of all the lines

Example Program 7.1.1 Sample Code for Array-Based Solution (main.cpp)

```

1  #include <vector>
2  using std::vector;
3
4  #include <string>
5  using std::string;
6
7  #include <sstream>
8  using std::ostringstream;
9
10 #include "simpleio.h"
11
12 // prototypes
13 void ComputeScale (const vector<float>&, const vector<float>&,
14                     const int, float&);
15 void GraphCoordinates (const float, const float, const float,
16                        float&, float&);
17 void Move (const float, const float);
18 void Draw (const float, const float);
19
20 int main ()
21 {
22     int nPoints;      // total number of points
23     int nLines;       // total number of lines
24
25     // obtain the number of points
26     do {
27         GetInteractive ("Total number of points: ", nPoints);
28     } while (nPoints <= 1);
29
30     // obtain the number of lines
31     do {
32         GetInteractive ("Total number of lines: ", nLines);
33     } while (nLines <= 0);
34
35     // storage for nodal coordinates
36     vector<float> fVX(nPoints), fVY(nPoints);
37
38     // obtain the nodal coordinates
39     for (int i=1; i <= nPoints; i++) {
40         {
41             ostringstream szPrompt;
42             szPrompt << "X coordinate for point " << i << ":" ;
43             GetInteractive (szPrompt.str(), fVX[i-1]);
44         }
45         {
46             ostringstream szPrompt;
47             szPrompt << "Y coordinate for point " << i << ":" ;
48             GetInteractive (szPrompt.str(), fVY[i-1]);
49         }
50     }
51
52     // storage for line start and end points
53     vector<int> nVSP(nLines), nVEP(nLines);
54
55     // obtain the start and end point numbers

```

```

56         for (i=1; i <= nLines; i++) {
57             int nTemp;
58             do {
59                 ostringstream szPrompt;
60                 szPrompt << "Start point for line " << i << ": ";
61                 GetInteractive (szPrompt.str(), nTemp);
62             } while (nTemp < 1 || nTemp > nPoints);
63             nVSP[i-1] = nTemp;
64             do {
65                 ostringstream szPrompt;
66                 szPrompt << "End point for line " << i << ": ";
67                 GetInteractive (szPrompt.str(), nTemp);
68             } while (nTemp < 1 || nTemp > nPoints);
69             nVEP[i-1] = nTemp;
70         }
71
72         // compute the scaling factor, fSF
73         float fSF;
74         ComputeScale (fVX, fVY, nPoints, fSF);
75
76         // now draw
77         for (i=1; i <= nLines; i++) {
78             float fXgs, fYgs;
79
80             // move to start point of line
81             int nS = nVSP[i-1];
82             // compute graph coordinates
83             GraphCoordinates (fVX[nS-1], fVY[nS-1], fSF, fXgs, fYgs);
84             Move (fXgs, fYgs);
85
86             // end point of line
87             int nE = nVEP[i-1];
88             GraphCoordinates (fVX[nE-1], fVY[nE-1], fSF, fXgs, fYgs);
89             Draw (fXgs, fYgs);
90         }
91
92     return 0;
93 }
```

Since we have not defined the details of the drawing-related functions – scaling, computing the graph coordinates, move and draw, we will simply provide their prototypes as shown in lines 13-18. Recall how C++ treats variables and functions – they must be defined before they can be used. We will store the vectors in C++’s `vector` container. A container, as the name suggests, not only provides storage space to store different data types but also provides operations that make data manipulation easy to implement. There are other types of containers such as list, deque, set, map etc. The `vector` class is used to store the four vectors as shown in lines 36 and 53.

As we have seen before, the values of the point coordinates and the start and end points of lines are obtained interactively using the `GetInteractive` function. Once the input data is in place, the scale factor is computed (Step 3 of the algorithm) as seen in line 74 using a call to (as yet undeveloped) function `ComputeScale`. Steps 4 through 7 are implemented in lines 77 through 90. Three other undeveloped functions `GraphCoordinates`, `Move` and `Draw` are used to compute the graph coordinates given the point coordinates, move to a location on the graph, and draw to a location on the

graph. One can imagine the `Move` function as moving to a point on the graph with the pen up (or, raised) and the `Draw` function as moving to a point on the graph with the pen down.

Strengths of the Array-Based Solution

- Easy to understand, visualize and code.
- Vector data structure requires minimal execution time and storage.

Weaknesses of the Array-Based Solution

- Typical engineering programs have tens if not hundreds of arrays. The management of all the arrays can be problematic.
- Extension of the program from two to three-dimensions by the simple addition of z coordinate values is likely to cause ripple effects throughout the program. Line 36 needs to be changed. The block of three lines to obtain the z-coordinate needs to be added. The z coordinate values need to be considered in the scaling factor function. Similarly, the `GraphCoordinates` function needs to be changed¹.
- Similarly, if newer objects such as curved beams (described by three points not two), are brought into the program, the programmed drawing logic needs drastic changes.
- Finally, for engineering applications, it is awkward to count starting at zero. This is one of the drawbacks of the `vector` class.

Data Abstraction

As program logic becomes more complex, it is certainly helpful to break the logic into smaller pieces. These smaller pieces are typically implemented in well-defined functions. Modularization in program development also has other beneficial effects – testing and debugging of smaller components is much easier, several different programmers can work simultaneously on a project with minimal interaction or overlap, and program reuse is possible. However, as the complexity of data flow in a program increases, the task of program development and maintenance can become expensive and fraught with dangers. It is now necessary to think about program development in a completely different way. This does not imply that whatever we have learnt in the preceding chapters is incorrect or useless. As we will learn in this chapter, it is easy to build on what we have learnt by simply reorganizing our thought process and learning new language constructs.

Ideally, programmers would like to define their own data types depending on the type of application program. These data types would be built using the C++ standard data types. The thought process to create these user-defined types is known as data abstraction. Data abstraction is defined as separating the overall properties of a data type from its implementation. The mechanism to implement the data abstraction is called encapsulation.

¹ The process of transforming a three-dimensional object to an equivalent view in a two-dimensional plane involves several more steps than just scaling and translation.

Let us look at a simple example having to store and manipulate information dealing with points defined in a two-dimensional space. Let this space be defined as the usual cartesian x-y space. Each point will be defined in terms of its x and y coordinates. To store the two coordinates, we will use the standard `float` type. Recall from Chapter 1 that objects are identified by a name, have *attributes* (defined as having properties) and *behavior* (capabilities to do something). The attributes of the point object are its (x,y) coordinates. What are some of the data manipulation that we may want to carry out with points in a two-dimensional space? For example, we may want to store and retrieve these values. We may also want to check whether the point is at the origin of the coordinate system, to compute what is the distance from this point to another point, etc. These are the type of information that a programmer needs to know to write a program that uses the point-related information. Such a code (that uses the point information) is called the client code. The client code needs to know how to use the information but not how the information is stored or how the functionalities (behavior) are implemented. Encapsulation also referred to information hiding, refers to the technique by which data attributes and behavior-related operations are linked together such that the data can be manipulated only through these operations not directly. The program or code that stores the data and implements the behavior is called the server code.

In C++ terminology, point (or whatever name we assign) is a class. This class is a user-defined data type similar to the built-in data types that C++ provides as `int`, `float`, etc. We associate objects with a class similar to the way we have been associating variables to the built-in data types. The client code declares objects associated with classes and uses the operations permitted by the class definition and implemented in the server code, to manipulate the information stored in the objects. One can now appreciate why information hiding is useful. First, the programming task can be very nicely divided. Programmers who have the knowledge and expertise in writing the client code can concentrate on getting their job done without having to worry about the implementation details. These are left to the experts who know more about the class attributes and its behavior. Second, the client code cannot inadvertently or otherwise make errors in setting or changing the values of the data. For example, if the point class is designed for points in the positive (x,y) space, then the error detection can be easily implemented in the server code and an appropriate action can be taken. Third, by separating the behavior from implementation, we minimize the impact that changes in behavior would have in the maintenance of a program. Let's go back to the point example. Let's assume that users of the class now request that they would like to look at points in a cylindrical coordinate system through two attributes (r, θ) . Does this imply that we would have to rewrite the existing client code or can we make changes to the server code (point class) such that this new functionality is available without comprising on the existing functionalities?

In the rest of the chapter, we explain the “how and why” of object-oriented programming using C++.

7.2 Components of a Class

What is the difference between a class and an object? *An instance of a class is an object.* Hence we have one and only one class definition in a program and we could have several objects that are different instances of that class.

The details of class definitions are presented next.

7.2.1 Defining Classes

In C++, the simplest class definition is shown below.

```
class class_name {
    public:
    ...
    private:
    ...
};
```

A class can have public and private member variables and functions². The public variables and functions are available outside of the class. The private variables and functions cannot be accessed directly outside of the class. Note that a complete class definition requires that the member functions and variables be defined within the `{ };` including the semicolon at the end.

We will go back to the *point class* discussed before. To manipulate the coordinate values, we will need member functions (functions that are members of a class) that will define, redefine and access the coordinate values. Function or functions that help define the values are called constructors (ctor for short). Similarly, the functions to redefine (or modify) the values are called modifier (or mutator) functions, and the functions to access the values are called accessor functions. Here are the statements to define this class. These statements are usually contained in a header file, say `point.h`.

```
#ifndef __RAJAN_POINT_H__
#define __RAJAN_POINT_H__

#include <string>

class CPoint
{
public:
    // constructors
    CPoint ();           // default
    CPoint (float, float); // overloaded
    // helper function
    void Display (const std::string&);
    // modifier function
    void SetValues (float, float);
    // accessor function
    void GetValues (float&, float&);

private:
    float m_fXCoor;    // stores x-coordinate
    float m_fYCoor;    // stores y-coordinate
};

#endif
```

² There is a third type – `protected` that we will see later.

Constructors are special member functions that do **not** require a return type. They have the same name as the class name. They are automatically called when an object associated with the class is declared and created. They can be overloaded just as regular functions. They are optional. If you do not define a constructor in your class definition, then C++ provides a constructor that essentially does nothing as far as your specifications are concerned. Constructors are typically used to initialize the members of the class. There is another special member function called the destructor (dtor for short). The destructor has the same name as the class, does not have a return type, cannot be overloaded, is not required to be declared or defined, and is automatically invoked when the object associated with the class goes out of scope. As an example, the destructor for the CPoint class can be declared as follows.

```
public:
    // constructors
    CPoint ();
    CPoint (float, float); // overloaded
    // destructor
    ~CPoint ();
```

The `~` (tilde) symbol is used before the class name to denote the destructor. The default specification for member variables and functions is private. In other words, if the keyword `public` or `private` is not used, the compiler assumes that the member function or member variable is private. In the CPoint class, two constructors are used. The first (without any parameters) is called the default constructor. We will use this constructor to set both the coordinate values to zero. In addition to the two constructors, there are three public member functions. The `Display` function is designed to display the coordinates using standard output. The `SetValues` and the `GetValues` functions are designed to redefine the coordinates and to obtain the coordinates, respectively. There are no public member variables. As we saw with data abstraction, class declarations typically do not have public member variables. The two variables that store the coordinates are declared as private variables. They cannot be accessed in any program component outside of the five member functions in the CPoint class. We could create exceptions to this rule as we will see in Chapter 9. There are no private member functions in the CPoint class declaration. The keywords `public` and `private` appear only in the class definition.

To implement (or define) these functions, we need to create the appropriate C++ statements. Here are those statements (usually contained in a source file, say `point.cpp`).

```
#include <iostream>
#include <string>
#include "point.h"

// default constructor
CPoint::CPoint ()
{
    // coordinates initialized to zero
    m_fXCoor = 0.0f;
    m_fYCoor = 0.0f;
}

// constructor
CPoint::CPoint (float fx, float fy)
{
```

```

    // coordinates set to fx and fy
    m_fXCoor = fx;
    m_fYCoor = fy;
}

// modifier function
void CPoint::SetValues (float fx, float fy)
{
    // coordinates set to fx and fy
    m_fXCoor = fx;
    m_fYCoor = fy;
}

// accessor function
void CPoint::GetValues (float& fx, float& fy)
{
    // coordinates returned in fx and fy
    fx = m_fXCoor;
    fy = m_fYCoor;
}

// helper function
void CPoint::Display (const std::string& szBanner)
{
    // display the current coordinates
    std::cout << szBanner
        << "[X,Y] Coordinates = [ "
        << m_fXCoor << ","
        << m_fYCoor << "].\n";
}

```

Note the difference between the definition of a regular function and a member function that belongs to a class. The member function definition needs a qualifier - the name of the class and the scope operator `::`. The statements in each member function are just as they would appear in any function with the difference that the member variables are declared in the class definition and hence should **not** be defined in the body of the member function. It would be **incorrect** to write the function `SetValues` as follows.

```

void CPoint::SetValues (const double dX, const double dY)
{
    double m_dXCoordinate; // local to this function!
    double m_dYCoordinate; // local to this function!
    m_dXCoordinate = dX;
    m_dYCoordinate = dY;
}

```

With the (incorrectly defined) function shown above, the values of `dX` and `dY` are assigned to the local variables `m_dXCoordinate` and `m_dYCoordinate` not the variables (with the same name) that are a part of the `CPoint` class! Recall the scope rules from Chapter 4.

The next obvious question is how can the class be used in an application program? We illustrate the usage using a simple example.

Example Program 7.2.1 Using the `CPoint` class

Here is the main program that illustrates the usage of the CPoint class. We will use two objects associated with the CPoint class – Origin and CarCoords. Origin will be declared using the default constructor and CarCoords will use the overloaded constructor. The user will be prompted to enter the x, y coordinates and these values will be used with the SetValues member function to set the user-defined coordinate values. The verification of the coordinate values will take place using the Display member function. The usage of the GetValues function is left as an exercise.

main.cpp

```

1  #include <iostream>
2
3  #include "point.h"
4  #include "..\library\getinteractive.h"
5
6  int main ()
7  {
8      // declaration invokes the default constructor
9      CPoint Origin;
10     // declaration invokes the overloaded constructor
11     CPoint CarCoords (0.0f, 0.0f);
12
13     // display the coordinates of the origin
14     CarCoords.Display ("Origin ");
15
16     // get the coordinates from the user
17     float fV[2];
18     GetInteractive ("Enter car coordinates: ", fV, 2);
19     // set the values via public member function SetValues
20     CarCoords.SetValues (fV[0], fV[1]);
21
22     // display the values
23     CarCoords.Display ("Car ");
24
25     return 0;
26 }
```

An object is declared just as any other variable in a program. In line 9, the object `Origin` is declared, and the coordinate values are initialized using the default constructor. It should be noted that this statement will not compile if a default constructor is not defined. In line 11, the object `CarCoords` is declared and the coordinate values are initialized using the overloaded constructor. Note how the member functions are used in the program. In line 14, the `Display` member function is called. The general usage in using a member function is

object.memberfunction (parameter list);
not

memberfunction (parameter list);

The `.` is referred to as the **member selection operator** and is used in accessing the member variables and member functions outside the class. Every object is closely tied to the variables associated with the class. In other words, an individualized copy of the member variables is created for every object.

However, only one copy of the member function is created that can then be used by all the objects associated with that class.

The use of private variables precludes its access outside the class. For example, we **cannot** write the following statements in the `main` function

```
CarCoords.m_fXCoor = fV[0]; CarCoords.m_fYCoor = fV[1];
```

instead of the original statement (line 20)

```
CarCoords.SetValues (fV[0], fV[1]);
```

Similarly, private member functions cannot be accessed outside the class.

Let us review the important facts about class definitions and usage.

1. The class definition is usually contained in a header file. A complete class definition requires that the member functions and variables be defined within the `{ };` including the semicolon at the end.
2. Member variables and functions are, by default, private.
3. Definition of the constructor is optional. C++ defines a constructor if one is not defined. The constructor does not have a return type and has the same name as the name of the class.
4. It is a good idea to define a default constructor.
5. Declaring a public member variable should be done with care. Unless the design of the class calls for a public member variable, declare all member variables as private. Private member functions and variables cannot be accessed outside of the class.
6. Member functions are defined using the scope resolution operator `::` and they are referenced outside the class using `.` the member selection operator.

We will now look at another example where (a) public and private member functions are used, and (b) some rudimentary error trapping is done via the **assert** library function.

assert library function

C++ provides library function `assert` to help in debugging a program. This function has the following prototype.

```
void assert (int expression);
```

If the expression is evaluated to be **false**³, the function prints a diagnostic error message and terminates the program. The intent is to use this function in trapping logical errors during the debugging phase of a program. The use of this function requires using the `<cassert>` header file. Once the program has been debugged, the assertion check can be tuned off (without changing the source code) using preprocessor directive.

```
#define NDEBUG           // NDEBUG stands for no debug
#include <cassert>
```

With `NDEBUG` defined, all calls to the `assert` function are ignored.

Example Program 7.2.2 A time-related class

We will look at defining and using a time-related class. This class is described in terms of three private member variables - hours, minutes and seconds. The client code will have access to these member variables via accessor and modifier functions. The accessor functions will provide access to either the hour, or the minute, or the second, or all the three member variables. Similarly, for the modifier functions. In addition, we will also have publicly available helper functions that will print the current time and also compute the elapsed time between two given time values.

The header file that contains the class declaration is shown next.

time.h

```
1  #ifndef __RAJAN__TIME_H__
2  #define __RAJAN__TIME_H__
3
4  Class CTime {
5      public:
6          CTime ();                  // default constructor
7          CTime (int, int, int);    // constructor
8          ~CTime ();                // destructor
9
10         // helper functions
11         void Print (int nOrder=0);
12         void TimeDifference (CTime, CTime);
13
14         // modifier functions
15         int SetTime (int, int, int);
16         int SetHour (int);
17         int SetMinute (int);
18         int SetSecond (int);
19
20         // accessor functions
21         void GetTime (int&, int&, int&);
22         int GetHour ();
23         int GetMinute ();
24         int GetSecond ();
```

³ C++ treats a positive integer value as true.

```

26     private: // store in 24-hour format
27         int m_nHour;
28         int m_nMinute;
29         int m_nSecond;
30
31     // helper functions
32     int ConvertToSeconds();
33     void ConvertFromSeconds (int, int&, int&, int&);
34 };
35
36 #endif

```

Internally the time is stored in a 24-hour format. To help in computing the time difference, we will also define and use two private helper functions that will convert the time from the 24-hour format to seconds and back.

The implementation of the member functions is shown next.

time.cpp

```

1  #include <cassert>
2  #include <iostream>
3  #include <iomanip>
4  #include "time.h"
5
6  // default constructor
7  CTime::CTime ()
8  {
9      m_nHour = 0;      // midnight
10     m_nMinute = 0;
11     m_nSecond = 0;
12 }
13
14 CTime::CTime (int nH, int nM, int nS)
15 {
16     // valid input?
17     assert (nH >= 0 && nH <= 23);
18     assert (nM >= 0 && nM <= 59);
19     assert (nS >= 0 && nS <= 59);
20
21     // set the values
22     m_nHour = nH;
23     m_nMinute = nM;
24     m_nSecond = nS;
25 }
26
27 // destructor
28 CTime::~CTime ()
29 {
30 }
31
32 // helper functions
33 void CTime::Print (int nOrder)
34 {
35     if (nOrder == 0)
36         std::cout << std::setw(2) << m_nHour << ":"
```

```

37             << std::setw(2) << m_nMinute << ":"  

38             << std::setw(2) << m_nSecond;  

39     else  

40         std::cout << std::setw(2) << m_nHour   << " Hour(s) "  

41         << std::setw(2) << m_nMinute << " Minute(s) "  

42         << std::setw(2) << m_nSecond << " Seconds(s) " ;  

43     }  

44  

45 void CTime::TimeDifference (CTime TFrom, CTime TTo)  

46 {  

47     // find the difference between the two times in seconds  

48     int nTFrom = TFrom.ConvertToSeconds();  

49     int nTTo = TTo.ConvertToSeconds();  

50  

51     int nDiff = nTTo - nTFrom;  

52     // adjust if time crosses midnight  

53     if (nDiff < 0)  

54         nDiff = nDiff + 86400;  

55  

56     // now convert from seconds back to hr:min:s  

57     ConvertFromSeconds (nDiff, m_nHour, m_nMinute, m_nSecond);  

58 }  

59  

60 int CTime::ConvertToSeconds ()  

61 {  

62     return (m_nHour*3600 + m_nMinute*60 + m_nSecond);  

63 }  

64  

65 void CTime::ConvertFromSeconds (int nTime, int& nH, int &nM, int& nS)  

66 {  

67     nH = (nTime/3600);  

68     nTime = nTime - nH*3600;  

69     nM = (nTime/60);  

70     nTime = nTime - nM*60;  

71     nS = nTime;  

72 }  

73  

74 // accessor functions  

75 void CTime::GetTime (int& nH, int& nM, int& nS)  

76 {  

77     nH = m_nHour;  

78     nM = m_nMinute;  

79     nS = m_nSecond;  

80 }  

81  

82 int CTime::GetHour ()  

83 {  

84     return (m_nHour);  

85 }  

86  

87 int CTime::GetMinute ()  

88 {  

89     return (m_nMinute);  

90 }  

91  

92 int CTime::GetSecond ()  

93 {

```

```

94         return (m_nSecond);
95     }
96
97 // modifier functions
98 int CTime::SetTime (int nH, int nM, int nS)
99 {
100     // check for valid values
101     if (nH < 0 || nH > 23) return(1);
102     if (nM < 0 || nM > 59) return(1);
103     if (nS < 0 || nS > 59) return(1);
104
105     // set the values
106     m_nHour = nH;
107     m_nMinute = nM;
108     m_nSecond = nS;
109     return (0);
110 }
111
112 int CTime::SetHour (int nH)
113 {
114     if (nH < 0 || nH > 23) return(1);
115
116     m_nHour = nH;
117     return (0);
118 }
119
120 int CTime::SetMinute (int nM)
121 {
122     if (nM < 0 || nM > 59) return(1);
123
124     m_nMinute = nM;
125     return (0);
126 }
127
128 int CTime::SetSecond (int nS)
129 {
130     if (nS < 0 || nS > 59) return(1);
131
132     m_nSecond = nS;
133     return (0);
134 }
```

The default constructor is listed in lines 7 through 12. The default time is set as midnight. The overloaded constructor has three parameters for the hour, minute and seconds. The constructor checks the input for errors in lines 17-19. An assertion failure occurs if the hour, minute or second has an invalid value such that any of the expressions is false. This is carried out to ensure that the time value is valid when the time object is created. Fig. 7.2.1 shows a sample error message.

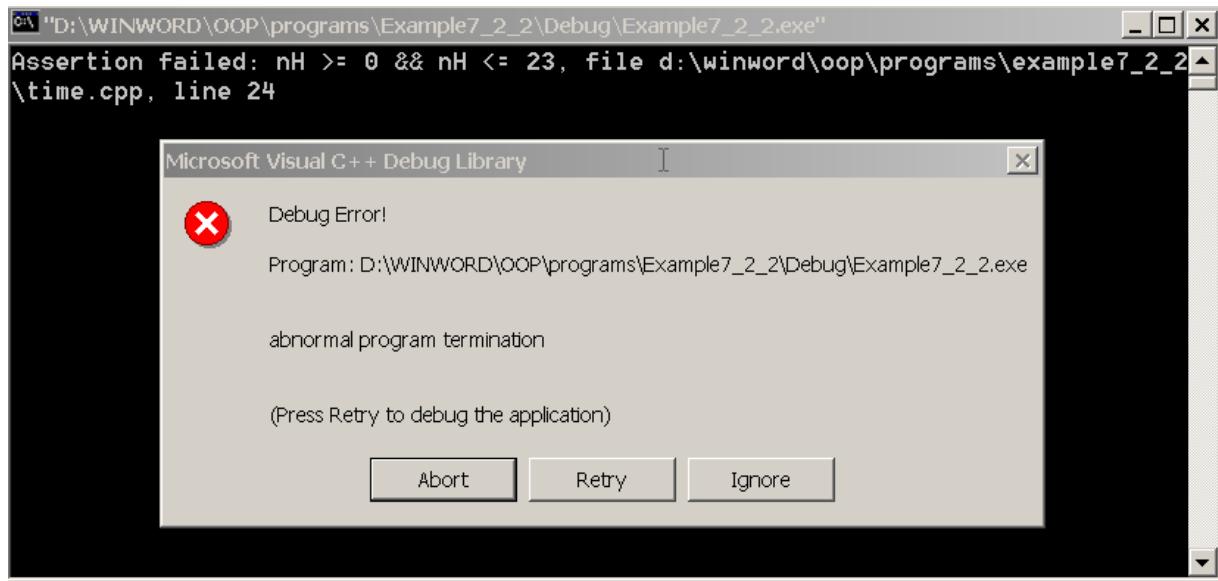


Fig. 7.2.1 Assertion failure error message from Microsoft Visual C++

The destructor has no statements and can be deleted both from the header and the source files. However, it is recommended that the destructor be included with every class just in case the destructor may be needed for future enhancements to the class. The `Print` member function prints the time as *hour:minute:second* (if the `nOrder` parameter has a zero value) or as *hour Hour(s)* *minute Minutes(s)* *second Second(s)*. There are four modifier functions (lines 97-134) that can be used to set or change the three time attributes. Once again, error checks are carried out to ensure that the specified values are valid. The return value of zero signifies a successful set operation; a nonzero value indicates that the specified value is invalid. Similarly, the accessor functions (lines 74-95) can be used to obtain or access the time values. The `TimeDifference` member function illustrates how private member functions can be used. In lines 48 and 49, the start time (or from time) and the end time (or to time) are converted from the 24 hour values into seconds elapsed since midnight. If the two time values span midnight, then the adjustment is made in line 54. Finally, in line 57, the time difference in seconds is converted back to the 24-hour format. The private member functions (`ConvertToSeconds` and `ConvertFromSeconds`) are defined in lines 60 through 72. An important question is how do lines 48 and 49 work since the `ConvertToSeconds` is a private member function? When a parameter of a class member function is of the same class type, the function can access the parameter's private member components (instead of using the accessor functions). In this example, the objects `TFrom` and `TTo` are both `CTime` objects. Hence it is legal to use `TFrom.ConvertToSeconds()` and `TTo.ConvertToSeconds()`.

We will now see a sample main program, contained in file `main.cpp`, which uses the `time` class.

main.cpp

```

1 #include <iostream>
2
3 #include "time.h"

```

```

4   #include "..\library\getinteractive.h"
5
6   int main ()
7   {
8       // declare variables (objects) to store start and end times
9       CTime StartTime;
10      CTime EndTime;
11      int nState;           // error indicator
12
13      // local variable to hold time data (hour, min and sec)
14      int nV[3];
15
16      // get start time
17      GetInteractive ("Enter start time. Hour, Minute and Second: ",
18                      nV, 3);
19      nState = StartTime.SetTime (nV[0], nV[1], nV[2]);
20      // valid time value?
21      if (nState != 0)
22          std::cout << "Invalid input. Check the values.\n";
23
24      // get end time
25      GetInteractive ("Enter end time. Hour, Minute and Second: ",
26                      nV, 3);
27      nState = EndTime.SetTime (nV[0], nV[1], nV[2]);
28      // valid time value?
29      if (nState != 0)
30          std::cout << "Invalid input. Check the values.\n";
31
32      // compute difference in time and display
33      CTime TimeDiff;
34      TimeDiff.TimeDifference (StartTime, EndTime);
35      std::cout << "\nTime difference between "; StartTime.Print ();
36      std::cout << "\n                                and "; EndTime.Print ();
37      std::cout << "\n                                is "; TimeDiff.Print (1);
38      std::cout << std::endl;
39
40      return 0;
41  }

```

The objects `StartTime` and `EndTime` are declared in lines 9 and 10 and are set as midnight since the default constructor is called. The overloaded constructor will be used if the declaration is as follows.

```
CTime StartTime (2, 15, 30);
```

Using the `GetInteractive` function, the start time and end time values are obtained. The modifier function `SetTime` is used to set the time values. An error message is displayed if this function returns a nonzero value.

In line 33, a new object `TimeDiff` is declared. This is the object that will store the time difference between `StartTime` and `EndTime`. The use of the `TimeDifference` function is illustrated in line 34. Finally, the `Print` member functions are used in lines 35 through 37 to display the results of the time computations.

Now that we have seen how to define and use simple classes, it is time for us to start formalizing this process.

7.3 Developing and Using Classes

There are much more to classes than we have seen in the previous section. In this section, we will look at a few of the advanced features that will be useful in the development of numerical analysis-based computer programs.

Defining functions in header files

Sometimes member functions in a class have very few statements, sometimes just one statement! In situations such as this, it is better to define the function in the class definition itself. Consider the CTime class from Section 7.2. We can define the body of the function `GetHour` in the header file as

```
int GetHour () {return m_nHour;};
```

Note that we need a semicolon at the end of the definition – a feature not required when the function is defined in the class source file. We can write the functions `GetMinute`, `GetSecond`, `SetHour`, `SetMinute` and `SetSecond` in a similar manner.

Inline Functions

As we saw in Chapter 4, functions make program development easier but at an added price of using additional resources of storage and execution time. C++ provides a mechanism by which this execution time overhead can be reduced through the use of inline functions identified by the `inline` keyword. For example, if the `GetHour` function is repeatedly used in a program, then a more efficient way of executing the function would be to declare it as

```
inline int GetHour () {return m_nHour;};
```

The `inline` qualifier is merely a suggestion to the compiler. The compiler is able to decide how best to optimize the function definition. One of the disadvantages of inlining a function is that the compiler inserts the same code at multiple locations where the function is called thereby making the executable code larger. The `inline` qualifier can also be used with regular (non-member) functions.

Using `const` Qualifier with Member Functions

As we have mentioned in the past, the `const` qualifier provides a safe mechanism in using and passing objects. For example, we used the `const` qualifier in the `CPoint` class with the `string` parameter in the `Display` member function. We can extend this safety mechanism when declaring and defining member functions. For example with the `CTime` class, we should have declared the `TimeDifference` member function as

```
void TimeDifference (const CTime&, const CTime&);
```

With this declaration, the two time objects that appear as the function parameters are passed as `const` references and hence cannot be modified within the class. When a function is declared as a `const` as follows

```
return_value class_name::function_name (parameter list) const;
```

then it cannot modify the object within the function body. Here is the modified CTime class definition that uses the `const` qualifier.

```
class CTime {
public:
    CTime ();           // default constructor
    CTime (int, int, int); // constructor
    ~CTime ();          // destructor

    // helper functions
    void Display (const std::string&, int nOrder=0) const;
    void TimeDifference (const CTime&);

    // modifier functions
    int SetTime (int, int, int);
    int SetTime (const CTime&);
    int SetHour (int);
    int SetMinute (int);
    int SetSecond (int);

    // accessor functions
    void GetTime (int&, int&, int&) const;
    void GetTime (CTime&) const;
    int GetHour () const;
    int GetMinute () const;
    int GetSecond () const;

private: // store in 24-hour format
    int m_nHour;
    int m_nMinute;
    int m_nSecond;

    // helper functions
    int ConvertToSeconds() const;
    void ConvertFromSeconds (int, int&, int&, int&) const;
};


```

Here is the definition of one of the `const` member functions.

```
int GetHour () const
{
    return m_nHour;
}
```

We have also made a simple improvement to one of the member functions. The `TimeDifference` member function now contains only one argument. The member function can be rewritten as follows.

```
void CTime::TimeDifference (const CTime& TTo)
{
    // find the difference between the two times in seconds
    int nTFrom = ConvertToSeconds();
    int nTTo = TTo.ConvertToSeconds();
    int nDiff = nTTo - nTFrom;
    // adjust if time crosses midnight
```

```

if (nDiff < 0)
    nDiff = nDiff + 86400;
// now convert from seconds back to hr:min:s
ConvertFromSeconds (nDiff, m_nHour, m_nMinute, m_nSecond);
}

```

It should be noted that while C++ does not allow the `const` qualifier to be used with constructors and destructors, `const` objects can be initialized using the constructor.

Composite Classes: Classes within Classes

As we have seen so far, class variables can be any of the standard C++ data types. In this chapter, we have started defining our own data types using the concept of classes. The class objects can be manipulated similar to the standard C++ data types. Hence a logical question is whether a class can contain objects belonging to other classes? Yes! When an object belonging to another class is declared as a member variable in another class, the object has a local scope within the defined class. When used correctly, one can build classes hierarchies that can have complex and useful functionalities. This process is known as class composition. We will illustrate the concepts associated with class composition in the following example. A class that has an object of other classes as the member variables is called a *composite class*.

Example Program 7.3.1 Example of class composition

Problem Statement: The City of Urban monitors temperature distribution in the city to help understand the “heat island” effect. A truck with the appropriate sensor-based electronic gear is sent out to obtain the temperature at different locations in the city at random times. Urban is divided into a grid and the precise location where the temperature reading is taken is obtained as an (x,y) pair. Develop a computer program to help the person collecting the data, enter and store the data suitable for processing at a later time.

Solution: We will defer the development of a complete solution that includes storing and retrieving the gathered data. In this example, however, we will develop the class declarations and definitions that will enable storage and retrieval at a later stage.

The problem statement identifies the following attributes with every temperature reading – the location in the form of (x,y) values, the time value and the temperature value. We have already defined the `CPoint` class to hold and manipulate data point-related information. We have also defined the `CTime` class to hold and manipulate time information. We could use these two classes in our solution. What is the advantage? The advantage is that this is a good example of class reuse. We will essentially leverage the usefulness of the two developed classes in defining a new class that will contain objects belonging to those two classes. For example, we can define three member variables as follows.

```

CTime m_TimeStamp;      // to store time temperature taken
CPoint m_Location;     // to store location temperature taken
float m_Temperature;   // to store the temperature value

```

The objects, `m_TimeStamp` and `m_Location` will be contained in the yet undefined class. What is the disadvantage? The disadvantage is that one needs to be familiar with the `CPoint` and `CTime`

classes to use them. In other words, instead of developing and learning about one class that would hold all the information we have now to deal with three classes.

The process of including objects from other classes into a class is known as class composition and the newly defined class is known as a composite class. In this problem, we will define a new (composite) class called `CSensor`. This class will contain the three member variables listed above. We will also define the usual accessor, modifier and help functions to manipulate the data. No additional features are necessary at this stage. The detailed algorithm is not required since the main program will merely get the sensor reading information from the user and display the information as a confirmation that the overall procedure is good.

The program uses slightly modified versions of previously defined classes – `CTime` and `CPoint`. The interested reader is urged to look at the source code to see what the changes are. The header file that contains the `CSensor` class declaration is shown next.

sensor.h

```

1  #include "time.h"
2  #include "point.h"
3
4  class CSensor {
5      public:
6          CSensor ();    // default constructor
7          ~CSensor ();   // destructor
8
9          // helper functions
10         void Display () const;
11
12         // modifier functions
13         void Set (const CTime&, const CPoint&, float);
14
15         // accessor functions
16         void Get (CTime&, CPoint&, float&) const;
17
18     private:
19         CTime      m_TimeStamp;        // time temperature taken
20         CPoint     m_Location;       // location temperature taken
21         float      m_fTemperature;   // temperature
22     };

```

There are three private member variables that store the three pieces of information associated with every reading. These variables are declared in lines 19-21. To manipulate the information, we use the usual modifier and accessor functions as shown in lines 13 and 16. The `Display` function is used to display the stored information. The class member functions are shown next.

sensor.cpp

```

1  #include <iostream>
2  #include <string>
3  #include "sensor.h"
4
5  // default constructor

```

```

6     CSensor::CSensor ()
7     {
8     }
9
10    // destructor
11   CSensor::~CSensor ()
12   {
13
14   }
15
16    // modifier function
17   void CSensor::Set (const CTime& Time, const CPoint& Point,
18                      float fTemp)
19   {
20       m_TimeStamp.SetTime (Time);
21       m_Location.SetValue (Point);
22       m_fTemperature = fTemp;
23   }
24
25    // accessor function
26   void CSensor::Get (CTime& Time, CPoint& Point,
27                      float& fTemp) const
28   {
29       m_TimeStamp.GetTime (Time);
30       m_Location.GetValues (Point);
31       fTemp = m_fTemperature;
32   }
33
34    // helper function
35   void CSensor::Display () const
36   {
37       std::cout << "Sensor Data ... \n";
38       m_Location.Display (" Location: ");
39       m_TimeStamp.Display (" Reading Time: ", 0);
40       std::cout << " Temperature: " << m_fTemperature << ".\n";
41   }

```

The Set and Get member functions are straightforward. Both use the publicly available functions from the CPoint and CTime classes to modify and access the data. The Display member function leverages the display functions from the CPoint and the CTime classes to display the entire sensor data – location, time and temperature.

Now we are ready to look at the main program used to obtain and store the information.

main.cpp

```

1  #include <iostream>
2
3  #include "sensor.h"
4  #include "..\library\getinteractive.h"
5
6  int main ()
7  {
8      const int NUMSENSORS = 4;
9      CSensor SensorData[NUMSENSORS]; // stores sensor data in a vector
10

```

```

11     // variables to generate the sensor data
12     int i;           // loop index
13     float fV[2];    // x-y coordinates
14     int nV[3];      // time data
15     float fTemp;    // temperature
16     CPoint Location; // to store location
17     CTime ReadingTime; // to store time of reading
18
19     // get the sensor data from the user
20     for (i=0; i < NUMSENSORS; i++)
21     {
22         // location, time and temperature
23         GetInteractive ("Sensor (x,y) location : ", fV, 2);
24         GetInteractive ("Time of reading (hr:min:sec) : ", nV, 3);
25         GetInteractive ("Temperature : ", fTemp);
26
27         // construct the location object
28         Location.SetValues (fV[0], fV[1]);
29
30         // construct the time object
31         ReadingTime.SetTime (nV[0], nV[1], nV[2]);
32
33         // set the sensor data
34         SensorData[i].Set (ReadingTime, Location, fTemp);
35     }
36
37     // display the sensor information
38     for (i=0; i < NUMSENSORS; i++)
39     {
40         SensorData[i].Display ();
41     }
42
43     // all done
44     return 0;
45 }
```

The first thing to notice is that a vector is used to store the sensor readings – line 9. For each of the sensors, the data is obtained interactively in lines 23-25. The `CPoint` object is created in line 28 and the `CTime` object is created in line 31. Finally, in line 34, the i^{th} `CSensor` object data is created. The newly created data is displayed one sensor at a time in line 40.

This simple example serves to illustrate what we have achieved. Instead of having one big class with several components we have three smaller classes with much smaller components and yet the same functionalities.

One has to be aware of the C++ rules that govern the initialization of objects contained inside other classes. For example the following (within `sensor.h`) is invalid and will not compile.

```

class CSensor
{
    ...
    CTime m_TimeStamp (10, 0, 0); // invalid initialization
    ...
}
```

These objects can be initialized only through an executable statement. For example, to initialize the `m_TimeStamp` variable we could modify the `CSensor` constructor as follows.

```
CSensor::CSensor ()
{
    m_TimeStamp.SetTime (10, 0, 0);
}
```

Another approach would be to overload the constructor.

```
CSensor::CSensor (const CTime& Time, const CPoint& Point)
{
    m_TimeStamp.SetTime (Time);
    m_Location.SetValues (Point);
}
```

Or, if appropriate **copy constructors** exist for the `CTime` and `CPoint` classes, then the following construct can be used.

```
CSensor::CSensor (const CTime& Time, const CPoint& Point) :
    m_TimeStamp (Time), m_Location(Point)
{}
```

Note that the `CSensor` constructor is called **after** all the data member class constructors (`CPoint` and `CTime`) are called. We will see more about composition in Chapter 9.

Copy Constructors

Copies of an object are made under the following conditions.

(a) When a declaration is made with *initialization from another object*. Here are three examples.

```
CPoint P1 (1.0f, 2.0f); // constructor.
CPoint P1 = P2;          // copy constructor.
CPoint P1 (P2);         // copy constructor.
```

(b) Parameters are passed by value.

(c) An object is returned by a function (more of that in Chapter 9).

C++ calls the *copy constructor* to make the copy. If there is no copy constructor defined for the class, C++ uses the default copy constructor that copies each field. The copy constructor takes a reference to a `const` parameter. It is `const` to guarantee that the copy constructor doesn't change it, and it is a reference parameter since a value parameter would require making a copy! Here is an example of a copy constructor for the `CPoint` class.

```
class CPoint
{
public:
    CPoint (const CPoint&);
    ...
}
```

```
CPoint::CPoint (const CPoint& P)      // copy constructor
{
    m_fXCoor = P.m_fXCoor;
    m_fYCoor = P.m_fYCoor;
}
```

7.4 Storage with `std::vector` class

At the beginning of this chapter we briefly saw a reference to the `std::vector`⁴ class. Now that we know more about classes, let us look at understanding how to use the standard `vector` class that is a container that can store and manipulate data in perhaps the most efficient fashion. It is an indexed container meaning that using an integer index, one can access elements of the vector. Note that `#include <vector>` must be used to make the proper reference to the `vector` class.

Declaration and Initialization

The following statement declares an `int` vector if size `n`.

```
std::vector<int> a(n); // assume that integer n > 0
```

The data type or object to be stored in the vector needs to be specified for a template class object to be created. The elements of the vector can be initialized as the following example shows.

```
std::vector<int> a(n, ival); // ival is an int variable
```

In the above example, all the `n` elements of vector `a` are initialized with the value contained in `ival`.

Assigning values to vector elements

The operator `[]` can be used to access elements of the vector. For example, the following code shows how all the elements of vector `a` are set to `ival`.

```
for (i=0; i < n; i++)
    a[i] = ival;
```

Finding the size of the vector

Insertion

Deletion

Accessors

⁴ `std::vector` is a template class that we will examine in sufficient detail in Chapter 9.

7.5 String manipulation with `std::string` class

Next we look at how to use the standard `string` class that provides a host of useful capabilities needed in manipulating strings. Firstly, we are able to treat character strings as a single entity, an object. Secondly, we do not have to guess the number of characters that the string might contain – the class automatically allocates the resources so that the string can grow or shrink on demand. Thirdly, there are functionalities that are provided through member functions that make string manipulation easy to carry out. In this section, we will look at the more important characteristics of the `string` class that will enable us to write useful programs. Note that `#include <string>` must be used to make the proper reference to the `string` class.

Declaration and Initialization

As we have seen earlier in the book, strings can be declared as a standard C++ data type. Here are some examples.

```
string szInput1, szInput2;      // empty strings
string szHeader = "Welcome to my world."; // initialized string
string szPrompt ("Input an integer: "); // initialized string
string szCpyPrt (szPrompt); // initialized string
```

Assigning values to strings

Once a string is declared, it can be populated just as a standard C++ type using the assignment operator and expressions. Here are some examples.

```
szName = szOldName;
szFullName = szFirstName + szLastName;           // concatenation
szFullName = szLastName + ", " + szFirstName; // concatenation
szFullName += szOldName;
```

The concatenation can also be carried out using the `append` member function.

```
szFullName = szFirstName;
szFullName.append (szLastName);
```

The “ ” pair is associated with the string class initialization even if the string contains a single character. In other words the following declaration is invalid

```
string szGradeA = 'A'; // invalid
```

but the following assignment

```
szGradeA = "A"; // valid assignment
```

is valid.

Accessing the individual components

The string class provides access to individual characters in a string through the use of [] operator similar to accessing an element of a vector.

```
szName = "Tony";
szName[3] = "i"; // name is now Toni
```

One can also find the length of a string using the `length` member function.

```
string szHeader = "Hello ";
std::cout << "Length of " << szHeader << " is "
<< szHeader.length() << "\n";
```

The above statements create the following display.

Length of Hello is 7

The member function `size` has the same functionality as the `length` member function.

Using as a function parameter

A string variable can be used as a function parameter just as any other variable. For example if the function prototype is

```
void Display (const string& szHeader, const float fV[], int nSize);
```

then the function can be called as

```
float fVA[10];
string szTitle = "Vector A";
Display (szTitle, fVA, 10);
```

If a string is to be modified by a function then it is preferable that it be passed as reference. An example usage in a program segment would be

```
string szFileName;
AddExtension (szFileName);
```

when the function prototype is

```
void AddExtension (string&);
```

Here is how the function `AddExtension` may be written.

```
void AddExtension (string& szFName)
{
    szFName += ".dat";
}
```

Comparing strings

Strings can be compared to each other using the logical operators (`==`, `!=`, `>`, `<`, `>=`, `<=`) as well as the `compare` member function. String comparisons take place in a lexicographical sense – as words are

arranged in a dictionary. For example, the word `list` precedes `listing`, `cooling` precedes `help` etc.

Here are a couple of examples of string comparisons.

```
if (szName1 < szName2)
    cout << szName1 << " occurs before " << szName2 << "\n";
else
    cout << szName2 << " occurs before " << szName1 << "\n";
```

Or

```
int nResult = szName1.compare(szName2);
if (nResult == 0)
    cout << szName1 << " is the same as " << szName2 << "\n";
else if (nResult < 0)
    cout << szName1 << " occurs before " << szName2 << "\n";
else if (nResult > 0)
    cout << szName2 << " occurs before " << szName1 << "\n";
```

Working with Substrings

The member function `substr` can be used to work with substrings. Let's look at the following example.

```
string szFirstPart, szLastPart;
string szTitle = "Vector A";
szFirstPart = szTitle.substr (0, 6); // extracts Vector
szLastPart = szTitle.substr (7, 1); // extracts A
```

The first parameter in the function call is the position from which to extract the substring and the second parameter is the number of characters to extract.

Finding substrings

Several member functions are provided that help find characters or strings within strings. Here are some of those member functions.

`find`: This function can be used to find a string within another string. The return value is the starting location where the string is found first (lowest position). Otherwise the returned value is a special value that is stored in `string::npos`. Here is an example.

```
string szHeadline = "Aliens land on Mars";
int nPos = szHeadline.find ("land");
if (nPos == string::npos)
    cout << "land is not contained in " << szHeadline << "\n";
else
    cout << "land occurs at location " << nPos <<
        " in string " << szHeadline << "\n";
```

With the above example, the function looks for the string "land". The returned value is 7.

find_first_of: This function can be used to find the first occurrence of any character in a given string within another string at or after a specified location. The default value of this location is 0. The return value is the starting location where the character is found. Here is an example.

```
string szHeadline = "Lose weight or loose change";
int nPos = szHeadline.find_first_of ("os", 5);
if (nPos == string::npos)
    cout << "could not find 'os' after location 5.";
else
    cout << "'os' occurs at location " << nPos << " beyond loc 5.\n";
```

With the above example, the function looks for the characters o and s beyond location 5. The returned value is 12 corresponding to the character o.

find_last_of: This function can be used to find the last occurrence of any character in a given string within another string at or before a specified location. The default value of this location is the end of the string. The return value is the location where the character is found. Here is an example.

```
string szHeadline = "Lose weight or loose change";
int nPos = szHeadline.find_last_of ("se");
if (nPos == string::npos)
    cout << "could not find se.";
else
    cout << "last se occurs at location " << nPos << ".\n";
```

With the above example, the function looks for the characters s and e from the end of the string. The returned value is 26 corresponding to the character e.

rfind: This function can be used to find a string within another string backwards. The return value is the location where the string is found last (highest position). Here is an example.

```
string szHeadline = "Aliens land on Mars";
int nPos = szHeadline.rfind ("land");
if (nPos == string::npos)
    cout << "land is not contained in " << szHeadline << "\n";
else
    cout << "land occurs at location " << nPos <<
        " in backward search of string " << szHeadline << "\n";
```

With the above example, the function looks for the string “land” within “Aliens land on Mars” but starts the search from the end of the string. The returned value is 7.

find_first_not_of: This function can be used to find the first occurrence (lowest position) at or after a specified location that matches none of the characters in a given string within another string. The default value of this location is 0. The return value is the starting location where the character is found. Here is an example.

```
string szHeadline = "xxxx wwwww xx xxxx change";
int nPos = szHeadline.find_first_not_of ("wx ", 5);
if (nPos == string::npos)
    std::cout << "could not find anything but 'wx ' after location 5.";
else
```

```
std::cout << "char not one of 'wx' occurs at location " << nPos
<< " after loc 5.\n";
```

With the above example, the function looks for the first character that is not one of w, x and a blank space beyond location 5. The returned value is 19 corresponding to the character c.

find_last_not_of: This function can be used to find the last occurrence (highest position) at or before a specified location that matches none of the characters in a given string within another string. The default value of this location is the end of the string. The return value is the starting location where the character is found. Here is an example.

```
string szHeadline = "xxxx wwwww xx xxxx change";
int nPos = szHeadline.find_last_not_of ("change ");
if (nPos == string::npos)
    std::cout << "could not find anything but 'change ' .";
else
    std::cout << "char not one of 'change ' occurs at location " << nPos
    << " string searched backwards.\n";
```

With the above example, the function looks for the first character that is not one of the characters in 'change' starting the search at the end of the string. The returned value is 17 corresponding to the character x.

Inserting strings

The **insert** member function can be used to insert a string at a specified location in another string. Here is an example.

```
string szDigits = "123321";
string szAlphabets = "abcd";
szDigits.insert (3, szAlphabets); // insert before location 3
```

The result is that szDigits is now 123abcd321.

Obtaining the address of a string

We will see more about pointers in the next chapter. When a situation requires the memory address of where the string is stored the **c_str** function can be used. We will see the usage of this function in the next example.

Case-related issues

Strings or components of a string can be manipulated with reference to upper and lower case representation of characters using C++'s character-handling library. The header file to include is **<cctype>**. The functions **toupper** and **tolower** convert a character to its upper case representation and lower case representation respectively, if the representation exists.

Here is a sample function that we can write to convert all the characters in a string to upper case using the **toupper** library function.

```
void ToUpperString (string& szInput)
{
    for (int i=0; i < szInput.length(); i++)
```

```

    {
        szInput[i] = toupper(szInput[i]);
    }
}

```

There are other functions available in the C++ library and these are listed and described in Appendix D.

Next we present an example program that uses the standard `string` class.

Example Program 7.5.1 Using the `std::string` class

We will rewrite the 4-function calculator developed in Section 4.6. We will improve the user input by obtaining an expression to evaluate rather than bits and pieces. The user will be prompted as

Input expression to evaluate:

However, the program will evaluate an expression in its simplest form as

leftnumber operator rightnumber

For example to evaluate 12×1.53 , the user is expected to enter the following expression.

Input expression to evaluate: **12*1.53**

We will use the following algorithm.

1. Obtain the expression from the user.
2. If the expression is `stop`, then terminate the program.
3. Look for `+-* /` within the expression. If one of the operators is not found, then the input expression is invalid.
4. Obtain the left number (from the beginning of the expression to the location of the operator). Convert from character string to a number.
5. Obtain the operator.
6. Obtain the right number (from the operator to the end of the expression). Convert from character string to a number.
7. Based on the operator carry out the operation between the two numbers and display the result.

The program that implements the above algorithm is presented below. Extensive use of the `string` library is made.

main.cpp

```
1 #include <iostream>
2 #include <string>
3 #include <math.h> // for atof function
4 #include "..\library\getinteractive.h"
5
6 int main ()
7 {
8     std::string szUserInput;
9     std::string szOper ("+-*/");
10    bool bError;
11
12    // loop until user input is stop
13    for (;;)
14    {
15        // initialize
16        bError = false;
17
18        // get user input
19        GetInteractive ("Input expression to evaluate: ", szUserInput, 50);
20
21        // terminate the program if input is "stop"
22        if (szUserInput == "stop")
23            break;
24        else
25        {
26            // find string length
27            int nLength = szUserInput.length();
28
29            // find the operator
30            int nPos = szUserInput.find_first_of (szOper, 0);
31            if (nPos > 0)
32            {
33                // obtain the first number
34                std::string szLeftNumber = szUserInput.substr (0, nPos);
35                double dLeftNumber = atof(szLeftNumber.c_str());
36
37                // operator
38                char szOperator = szUserInput[nPos];
39
40                // obtain the second number
41                std::string szRightNumber = szUserInput.substr (nPos+1, nLength);
42                double dRightNumber = atof(szRightNumber.c_str());
43
44                double dResult;
45
46                switch (szOperator)
47                {
48                    case '+': dResult = dLeftNumber + dRightNumber;
49                        break;
50                    case '-': dResult = dLeftNumber - dRightNumber;
51                        break;
52                    case '*': dResult = dLeftNumber * dRightNumber;
53                        break;
54                    case '/': dResult = dLeftNumber / dRightNumber;
55                        break;
56                }
57            }
58        }
59    }
60}
```

```

57         // display the result
58         std::cout << szUserInput << " = " << dResult << "\n";
59     }
60     else
61         std::cout << "Unable to find operator.\n";
62     }
63 }
64
65 return 0;
66 }
```

In line 9, the supported operators (or the four function-related operators) are defined and stored in the `szOper`s variable. Using the `find_first_of` member function, the location of the operator within the string is found in line 30. The string extraction to obtain the left and right numbers uses the `substr` member function (lines 33 and 40). To convert from a string to a floating point value, the `atof` function is used. As we can see in lines 34 and 41, the `c_str` member function is used to obtain the address of the string since the `atof` function requires a character pointer as the function parameter. The program does not check to see whether the character string is a valid number or not. In fact, the `atof` function returns a zero value if the input cannot be converted to a floating point number⁵.

There are other string related functionalities (such as stream processing) that we will see later in the book.

7.6 What is `struct` ?

Those familiar with the C language will recognize the `struct` keyword to define a structure that vaguely resembles a class. The syntax is as follows.

```

struct name_of_structure
{
    type variable1;
    type variable2;
    ...
};                                // note the semicolon
```

The above statement defines the (user-defined) type name that can then be used to define variables that have the defined structure. For example, we can define a point structure as follows.

```

struct stPoint
{
    float fXCoor;
    float fYCoor;
};
```

Using this definition, we can declare variables `CarCoordinates` and `BikeCoordinates` as follows.

⁵ The reader is urged to use the `GetLongValue` and `GetDoubleValue` functions available in the `getinteractive.cpp` file.

```
stPoint CarCoordinates, BikeCoordinates;
```

We can declare and initialize variables as follows.

```
stPoint CarCoordinates = {-55.1f, 0.0f};
```

```
stPoint CarCoordinates = BikeCoordinates;
```

To access the individual components of the structure, we will have to use the . (dot) operator. For example

```
CarCoordinates.fXCoor = 12.3f;
```

We can also equate one structure to another as

```
CarCoordinates = BikeCoordinates;
```

struct's provide a mechanism to aggregate data. However, they do not have the features that are found in classes.

7.7 Object-Oriented Solution

We will now develop an object-oriented solution to the problem posed in Section 7.1.

Object-based Solution

An examination of the problem statement shows that there are two entities – points and lines.

Attributes

Point	Line
Is identified by a point number.	Is identified by a line number.
Has an x-coordinate and a y-coordinate.	Defined in terms of two unique vertices or points – a start point and an end point.

Behavior

Point	Line
Obtain or define the x-coordinate, or the y-coordinate, or both.	Obtain or define the start point, or the end point, or both.

Compute the distance between two points.	Drawing process on a two-dimensional coordinate system.
Compute the maximum and minimum values.	

The following **bare** CPoint and CLine classes are proposed based on the above analysis (contained in point.h and line.h).

```
class CPoint
{
public:
    CPoint (float fX=0.0f, float fY=0.0f);
    CPoint (float fX, float fY, float fZ);
    ~CPoint ();
    // helper functions
    void Print ();
    // accessor functions
    void GetValues (float&, float&);
    void GetValues (float&, float&, float&);
    // modifier functions
    void SetValues (float, float);
    void SetValues (float, float, float);
private:
    float m_fXCoor;
    float m_fYCoor;
    float m_fZCoor;
};

class CLine
{
public:
    CLLine (int nSP=0, int nEP=0);
    ~CLLine ();
    // helper functions
    void Print ();
    // accessor functions
    void GetValues (int&, int&);
    // modifier functions
    void SetValues (int, int);
private:
    int m_nStartPoint;
    int m_nEndPoint;
};
```

The resulting main program is shown next. Note that when a vector object is passed to a function, it is passed as reference. If the object is passed as a value, then a local copy of the vector is made in the function. This is both time-consuming as well as resource-consuming. Passing by reference or const reference (see line 15) speeds up the execution of the program.

Example 7.7.1 Sample Code for Object-Based Solution (main.cpp)

```

1  #include <vector>
2  using std::vector;
3
4  #include <string>
5  using std::string;
6
7  #include <sstream>
8  using std::ostringstream;
9
10 #include "simpleio.h"
11 #include "point.h"
12 #include "line.h"
13
14 // prototypes
15 void ComputeScale (const vector<CPoint>&, const int, float&);
16 void GraphCoordinates (const float, const float, const float,
17                         float&, float&);
18 void Move (const float, const float);
19 void Draw (const float, const float);
20
21 int main ()
22 {
23     int nPoints;      // total number of points
24     int nLines;       // total number of lines
25
26     // obtain the number of points
27     do {
28         GetInteractive ("Total number of points: ", nPoints);
29     } while (nPoints <= 1);
30
31     // obtain the number of lines
32     do {
33         GetInteractive ("Total number of lines: ", nLines);
34     } while (nLines <= 0);
35
36     // vectors to store the point and line data
37     vector<CPoint> PointData(nPoints); // stores point data
38     vector<CLine> LineData(nLines);   // stores line data
39
40     // obtain the nodal coordinates
41     for (int i=1; i <= nPoints; i++) {
42         float fX, fY;
43         {
44             ostringstream szPrompt;
45             szPrompt << "X coordinate for point " << i << ": ";
46             GetInteractive (szPrompt.str(), fX);
47         }
48         {
49             ostringstream szPrompt;
50             szPrompt << "Y coordinate for point " << i << ": ";
51             GetInteractive (szPrompt.str(), fY);
52         }
53         PointData[i-1].SetValues (fX, fY);
54     }
55 }
```

```

56      // obtain the start and end point numbers
57      for (i=1; i <= nLines; i++) {
58          int nSP, nEP, nTemp;
59          do {
60              ostringstream szPrompt;
61              szPrompt << "Start point for line " << i << ":" ;
62              GetInteractive (szPrompt.str(), nTemp);
63          } while (nTemp < 1 || nTemp > nPoints);
64          nSP = nTemp;
65          do {
66              ostringstream szPrompt;
67              szPrompt << "End point for line " << i << ":" ;
68              GetInteractive (szPrompt.str(), nTemp);
69          } while (nTemp < 1 || nTemp > nPoints);
70          nEP = nTemp;
71          LineData[i-1].SetValues (nSP, nEP);
72      }
73
74      // compute the scaling factor, fSF
75      float fSF;
76      ComputeScale (PointData, nPoints, fSF);
77
78      // now draw
79      for (i=1; i <= nLines; i++) {
80          float fX, fY;
81          float fXgs, fYgs;
82          int nSP, nEP;
83
84          // get line information
85          LineData[i-1].GetValues (nSP, nEP);
86          // get start point coordinates
87          PointData[nSP-1].GetValues (fX, fY);
88          // compute graph coordinates
89          GraphCoordinates (fX, fY, fSF, fXgs, fYgs);
90          Move (fXgs, fYgs);
91
92          // get end point coordinates
93          PointData[nEP-1].GetValues (fX, fY);
94          // compute graph coordinates
95          GraphCoordinates (fX, fY, fSF, fXgs, fYgs);
96          Draw (fXgs, fYgs);
97      }
98
99      return 0;
100 }
```

The resulting program represents a small but significant improvement over the array-based solution.

Strengths of the Object-Based Solution

- Forces the program developer to think in terms of classes and objects. Abstraction and encapsulation makes program organization and development easier. This leads to cleaner data visualization and organization. There are less data management issues compared to the array-based solution.

- Function overloading permits us to think ahead with less programming repercussions.
- Vector data structure requires minimal execution time and storage.

Weaknesses of this Object-Based Solution

- The program organization still looks cluttered. The main program, for such a simple problem, should clearly spell out the major steps in the algorithm with the steps implemented in member functions.
- We still do not have a solution to how newer objects such as curved beams (described by three points not two), will be handled elegantly.
- Finally, for engineering applications, it is awkward to count starting at zero. This is one of the drawbacks of the **vector** class that we will overcome in the next section.

New and Improved Version

Improvements usually take place in small increments, and this is true of software development too. Here we present an improved version of the program. First we will define a **CWireFrame** class designed to hold all the attributes of the object being displayed.

```
class CWireFrame
{
public:
    CWireFrame ();
    ~CWireFrame ();
    // helper functions
    void ReadPointData ();
    void ReadLineData ();
    void ComputeScale ();
    void DrawModel ();
    void Print ();
    // modifier functions
    void SetSize (int, int);
    // accessor functions
    void GetSize (int&, int&);

private:
    int m_nPoints;
    int m_nLines;
    vector<CPoint> m_PointData;
    vector<CLine> m_LineData;
    float m_fxMin, m_fxMax; // x min and max coordinates
    float m_fyMin, m_fyMax; // y min and max coordinates
    float m_fSfx, m_fSfy; // scale factors, x and y directions
    void GraphCoordinates (int nP, float& fxgs, float& fygs);
    void Move (float fx, float fy);
    void Draw (float fx, float fy);
};

#endif
```

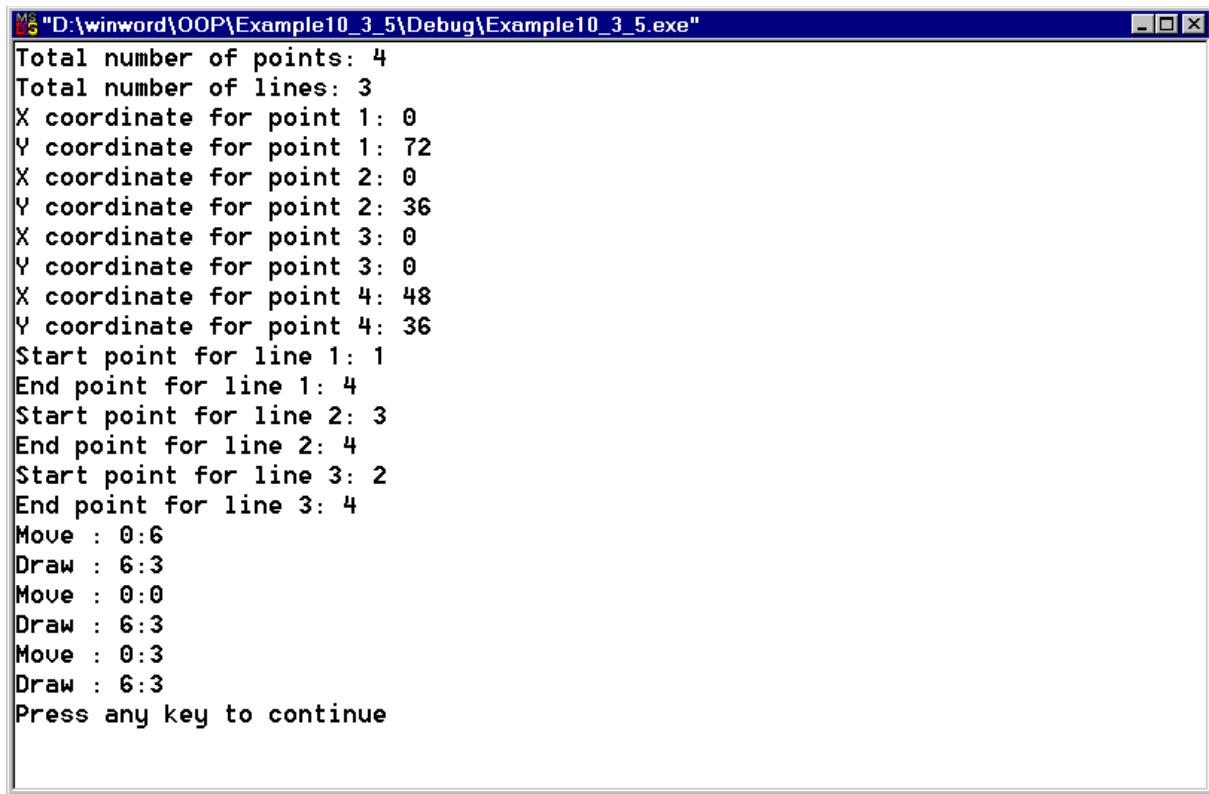
There is only one primary object in the program that is associated with the CWireFrame class. The list of private variables includes all the point and line data. The corresponding main program is as follows.

Example 7.7.2 Sample Code for the new and improved Object-Based Solution (main.cpp)

```

1  #include "simpleio.h"
2  #include "wireframe.h"
3
4  int main ()
5  {
6      CWireFrame Object; // the object
7      int nPoints, nLines;
8
9      // obtain the number of points
10     do {
11         GetInteractive ("Total number of points: ", nPoints);
12     } while (nPoints <= 1);
13
14     // obtain the number of lines
15     do {
16         GetInteractive ("Total number of lines: ", nLines);
17     } while (nLines <= 0);
18
19     // set problem size
20     Object.SetSize (nPoints, nLines);
21
22     // obtain the point information
23     Object.ReadPointData ();
24
25     // obtain the line information
26     Object.ReadLineData ();
27
28     // compute the scaling factors
29     Object.ComputeScale ();
30
31     // now draw
32     Object.DrawModel ();
33
34     return 0;
35 }
```

The variable `Object` contains the entire problem data and the algorithm steps are now implemented as member functions of the `CWireFrame` class. A sample execution of the program is shown next.



The screenshot shows a Microsoft Windows console window titled "D:\winword\OOP\Example10_3_5\Debug\Example10_3_5.exe". The window contains the following text output:

```
Total number of points: 4
Total number of lines: 3
X coordinate for point 1: 0
Y coordinate for point 1: 72
X coordinate for point 2: 0
Y coordinate for point 2: 36
X coordinate for point 3: 0
Y coordinate for point 3: 0
X coordinate for point 4: 48
Y coordinate for point 4: 36
Start point for line 1: 1
End point for line 1: 4
Start point for line 2: 3
End point for line 2: 4
Start point for line 3: 2
End point for line 3: 4
Move : 0:6
Draw : 6:3
Move : 0:0
Draw : 6:3
Move : 0:3
Draw : 6:3
Press any key to continue
```

Fig. 7.6.1 A sample truss description and program generated drawing instructions using the program from Example 7.6.2 (Microsoft Visual C++ Console Application)

While this is a much-improved solution, there are some subtle deficiencies. The `CLine` class should be the logical place to set up the draw function. However, the `CLine` class does not have direct access to the point's attributes such as the coordinates. The `CWireFrame` class is used to get both the line and associated point information.

Summary

In this chapter we were introduced to the concept and usage of simple yet powerful idea – how to visualize the logic and data encountered in a typical program through abstraction. This process helps in the defining classes where attributes are distinct from behavior. By using classes, we eliminate a host of problems. First, we tie data to data access and manipulation very tightly. We can make program development more systematic by reducing the proliferation of global functions and variables. Second, we can have more control over the access to the data through the use of public and private functionalities – information hiding is possible. Third, we encourage the reuse of software. This is possible only if classes are designed with appropriate specifications. As we will see in Chapter 13, programmers not involved in the initial design of a class can still add functionalities to an existing class through the process of inheritance.

Programming Style Tip 7.1: There is no substitute for proper class design

It is essential that the class design take place properly. The interface and the implementation should be separated - this is the idea behind encapsulation. From the client code development viewpoint, the programmer needs to know only the interface for a successful implementation.

Programming Style Tip 7.2: Practice defensive programming

Be careful to define `public` and `private` variables and functions. Functions that can potentially expose the inner workings of a call should be hidden from the client code by declaring them as `private`.

Programming Style Tip 7.3: Define a default constructor and a destructor

Even though C++ does not require a `ctor`, this is the place where one would initialize all the variables defined in a class. For similar reasons, it is better to explicitly define a destructor where cleanup operations can take place.

Programming Style Tip 7.4: Define the copy constructor

Even though C++ does not require a copy constructor, you can speed up program execution and avoid run time errors by providing a copy constructor.

EXERCISES

Most of the problems below involve the development of one or more classes. In each case develop (a) plan to test the classes(s), and (b) implement the plan in a [main](#) program.

Appetizers

Problem 7.1

Develop a class **CRectangle** to handle rectangles in a two-dimensional (X, Y) space. In addition to storing the data that describe the rectangle, this class should be able to (a) compute the area, (b) perimeter, and (c) recognize if the rectangle is a square.

Problem 7.2

Enhance the capabilities of the **CPoint** class by adding a member variable **m_fZCoor** (to store the z coordinate) and member functions to carry out the following tasks. (a) Create the copy constructor as **CPoint::CPoint (const CPoint&)**. (b) A predicate function **bool IsOrigin ()** to see if the point is at the origin of the coordinate system. (c) The distance to another point as **float DistanceTo (const CPoint&)**. (d) Unit vector to another point as **void UnitVector (const CPoint&, float fUVVector[])**.

Problem 7.3

The capabilities of the **CTime** class discussed in this chapter can be enhanced. Create additional member variables and functions that will (a) recognize the time zone, and (b) print time in 24-hour format, or in the am or pm format, or with respect to UTC (coordinated universal time; formerly known as Greenwich Mean Time).

Main Course

Problem 7.4

Develop a **CFraction** class to store fractions and support the following operations – addition, subtraction, multiplication and division via **Add**, **Subtract**, **Multiply**, and **Divide** member functions. The prototype of a typical public member function is as follows.

```
void Add (const CFraction&, const CFraction&);
```

Also, develop a **void Display (const std::string& szMessage);** public member function that will display the fraction in its reduced form preceded by the **std::string** argument.

Problem 7.5

Develop a **CTriangle** class as a composite class using the **CPoint** class object. The triangle is described in terms of the (x, y) coordinates of the three vertices. The class should store information

on the triangle such as perimeter, area and the three angles and have public accessor functions for these attributes. It should also have public predicate functions to test and see if the triangle is an isosceles (`bool IsIsosceles()`) triangle, right triangle (`bool IsRightTriangle()`) triangle or an equilateral (`bool IsEquilateral()`) triangle. Also, develop a `void Display(const std::string& szMessage);` public member function that will display the all the stored properties of the triangle.

Problem 7.6 (Chapter 6)

Develop a `CQuadraticPoly` class to find the roots of a quadratic polynomial $ax^2 + bx + c$. Construct the default and an overloaded constructor that accepts a, b, c . Store a, b, c as private member variables. Construct a public function `bool ComputeRoot (float&, float&)` where the return value is `true` if the roots are real and `false` if the roots are imaginary. The two parameters are the roots of the polynomial.

Problem 7.7 (Chapter 6)

TBC Numerical differentiation

Problem 7.8 (Chapter 6)

TBC Numerical integration

C++ Concepts

Problem 7.9

Develop two blueprints for a solid geometry program – one with and one without classes. The program should have the features to compute quantities such as length, interior angles, perimeter, area, surface area, and volume for the following objects – triangle, quadrilateral, tetrahedron, hexahedron, prism, pyramid, cylinder, and cube. Discuss the pro and cons of the two programs.

Chapter 8

Pointers

“A great memory does not make a mind, any more than a dictionary is a piece of literature.”

John Henry Newman

“Memory is the second thing to go”

General-purpose programs handle objects whose size is known only at run time and whose size may change dramatically during the course of execution. For example, a program that draws an X-Y graph is much less useful if it sets a predefined limit on the number of points it can handle or, if it does not allow addition or deletion of points. In this chapter and the next, we will see the basics of how to write programs where memory allocation to store scalars and arrays are handled dynamically at run time.

Nothing is free! The process of managing resources can be troublesome and can lead to unintended consequences. While the resources on any computer system are finite, programmers are being asked to create programs that can handle bigger problems, run faster and yield more accurate results.

Objectives

- To understand the concept associated with pointers.
- To understand more about dynamic memory allocation.
- To understand and practice writing C++ programs where memory allocation and deallocation are managed.

8.1 Memory Management

It is quite helpful to understand how memory is managed in a typical computer so that we can develop programs that manage memory resources intelligently. In this section, we present a simplified but nevertheless, useful view.

In a typical desktop computer or a workstation, information containing both data and instructions, flows through a number of hardware devices. Fig. 8.1 shows a schematic diagram. A computer program typically resides on a permanent storage device such as a (hard) disk. Program execution starts when the information from the computer program is loaded into the RAM via the disk I/O bus. From the RAM, information is passed onto the CPU for processing through the memory bus. The memory has its own hierarchy. There are registers in the CPU that store information. These are located closest to the logic units within the CPU where the information is processed. Cache memory forms the next level in the hierarchy. Physically, they are located either on the CPU chip or on a separate board and have a direct link to the CPU. There are different levels of cache identified as L1, L2 or even L3 cache. These are very high-speed memory (hence more expensive) but can store much less information than RAM. L1 cache is on the same chip as the microprocessor whereas the other cache memories are on a separate chip. Typically, the cache sizes vary from a few kilobytes (KB) to a few megabytes (MB) whereas RAM typically varies between a few MB to a few gigabytes (GB).

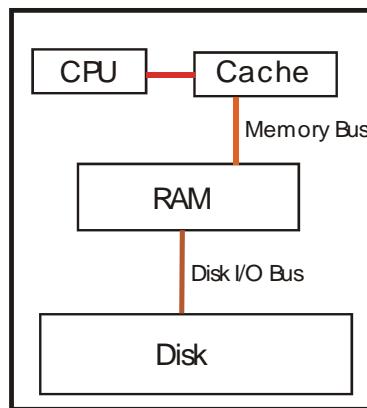


Fig. 8.1 Memory Hierarchy

The contents of the RAM can be thought of as being divided into two distinct parts – as used by the operating system and those used by one or more application programs.



Fig. 8.2 Memory Usage

Let's look at the simplest case where the amount of RAM is **greater** than the amount of memory required by the operating system and a single active application program. The program instructions and data are transferred from the RAM to the CPU on demand. What is the purpose of the cache? The CPU first looks to see whether the information it needs is in cache memory. If it is (cache hit), it fetches the information for processing. This may take a few nanoseconds. If the information is not in cache (cache miss), then the information must exist in the RAM and appropriate instructions are issued. This operation may take tens and hundreds of nanoseconds. One can see from this simple example, that everything else being the same, a program will run faster if there are more cache hits than misses. A program with more cache hits than another program, is said to have more *locality of reference*.

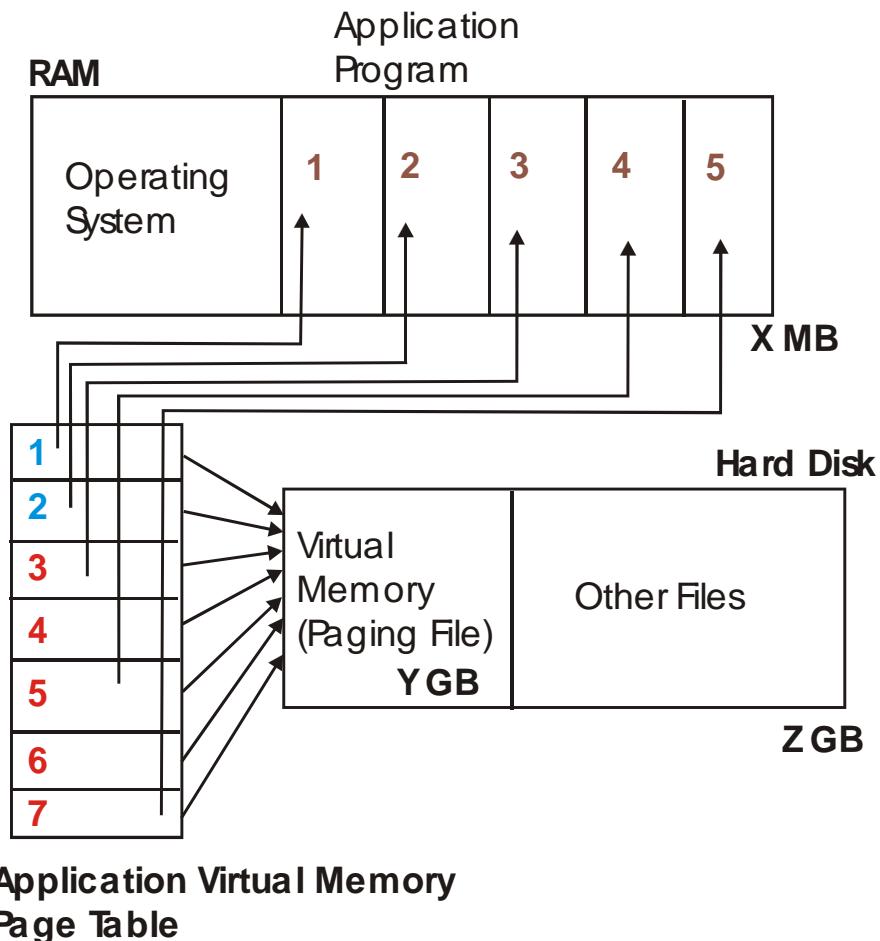


Fig. 8.3 A simplified virtual memory scenario

Let's look at the next case where the amount of RAM is **less** than the amount of memory required by the operating system and a single active application program. This situation is depicted in Fig. 8.3. An operating system that handles such a scenario is called a virtual memory operating system. Examples include Microsoft Windows NT/2000/XP, Linux, the different flavors of Unix, etc.

Conceptually, the memory is divided into pages. The size of a page (in bytes or KB or MB) is a function of the OS. In the figure, let us assume that after the entire operating system is loaded into RAM, five pages can be loaded into RAM. We will label these pages as 1, 2, ..., 5. Let us now assume that we wish to execute an application that requires a memory equivalent of 7 pages. Pages 1 and 2 are pages that contain program instructions, and the rest of the pages contain program data. In a virtual memory OS, a special part of hard disk is set aside for virtual memory related operations. Typically, the paging file size is much larger than the size of RAM. The purpose of the paging file is to maintain a copy of the program's information. This information is then made available to the CPU on demand from the disk to the CPU. This operation may take a few milliseconds to complete – three orders of magnitude slower than the amount of data transfer time from the RAM to the CPU. In the example, pages 1, 2, 3, 5 and 7 are called *swapped-in* pages since they are currently in RAM, and pages 4 and 6 are called *swapped-out* pages since they are currently not in RAM.

In order to maintain coherence between the different copies of program information, several strategies are implemented in a typical OS. One of the techniques is to use a virtual memory page table. This page table contains the mapping between the pages in the RAM and the location on the disk (part of the virtual memory paging file system) associated with the application. When an application issues a request for information, the system computes the memory address as a page number and checks to see if the page is in RAM. If it is (page hit), it fetches the information for processing. If the information is not in RAM (page miss), then the information must exist on the hard disk and appropriate instructions are issued. One can see from this simple example, that everything else being the same, a program will run faster if there are more page hits than misses. A program with more page hits than another program, is said to have more *locality of reference*.

Memory management becomes more complex as we begin to change the assumptions made with the previous examples. How is memory to be allocated and deallocated if these operations take place during the execution of the program? Most OS use *heap* or *free store* to carry out dynamic memory management. This is the memory area that gets affected due to the use of *new* and *delete* operations. We have *memory leak* if *new* is used without a corresponding *delete*. Similarly, we have *access violation*, if the program tries to access memory that is not allocated or refers to a non-existent memory location or address.

There are several types of objects and non-objects that a typical C++ program controls in different memory areas – **const** data, heap, free store, stack, and global and static memory areas [Herb Sutter].

const data: This is a special memory area that is protected and cannot be modified (read only). Only primitive data types whose values are known at compile time can be stored here. Objects cannot be stored. The data here is available for the entire duration of the program.

Stack: The stack is used to store the automatic variables. The objects are created as soon as memory is allocated and destroyed immediately before memory is deallocated.

Free store: This memory area is used for dynamic memory allocation and is affected by *new* and *delete* operators. Memory for objects is allocated but this memory may not be immediately initialized. This memory may be accessed and manipulated outside of the object's lifetime but while the memory is still allocated.

Heap: Heap is the second memory area used for dynamic memory allocation and is affected by `malloc` and `free` operators more commonly associated with C.

Global/static: Storage allocation for global or static variables takes place at program startup and the initialization takes place subsequently. For instance, a static variable in a function is initialized only the first time program execution passes through its definition.

Finally, it is helpful to understand what happens when objects are repeatedly created and destroyed on the free store or heap. Memory when used in this fashion, becomes fragmented. Unlike some other languages, C++ does not carry out *garbage collection*. Garbage collection is the process of recycling the memory space when that memory space is no longer needed. Programmers understanding and developing programs for numerical analysis must recognize that memory management is an important issue as are speed of execution and accuracy of results.

In the rest of the chapter, we will start looking at how to use free store during dynamic memory allocation.

8.2 Pointers

As people can locate places and things based on their addresses, pointers in C++ can be used to manipulate information based on memory addresses. Pointer data types represent a reference to an object or a location. Pointers may be specialized by the type of the object referred to. In this chapter, we will see pointers represented by a memory address; however, they can be more complicated as we will see in later chapters.

Here is a declaration of a pointer variable.

```
int *pnX;
```

The variable `pnX` can hold the memory address of a variable of type `int`. Note that the declaration requires an asterisk `*` in front of the variable name. The above style is preferable to the following.

```
int* pnX;
```

The problem occurs when multiple variables are declared with the same statement. For example, what is implied by the following statement?

```
int* pnX, pnY;
```

As it turns out, `pnX` is a pointer variable but `pnY` is a regular `int`. If the intent is to declare both of them as pointer variables, one must use the following declaration.

```
int *pnX, *pnY;
```

So how does one use a pointer variable? Consider the task of pointer variable `pnX` required to point to an `int` variable `nX`. One could generate the following statements.

```

int *pnX, nX;
nX = 10;
pnX = &nX;

```

In the first statement, variable pnX is declared as a pointer variable and nX is declared as an `int`. In the second statement, the value of nX is set as 10. In the third statement, the value (memory address) of pnX is set by using the address symbol `&` (this is the same symbol that we used when passing-by-reference) along with the variable that the pointer variable is pointing to. Let us now look at an example to learn more about pointers.

Example Program 8.2.1 Pointer example usage

In this example we will see the use of `int` variables and `int` pointers. The basic ideas can be extended to other data types. Simple pointer arithmetic is also illustrated

main.cpp

```

1  #include <iostream>
2
3  void ShowValues (int nV, int *pnV)
4  {
5      std::cout << "\n";
6      std::cout << "          nV is " << nV << "\n";
7      std::cout << "          pnV is (address) " << pnV << "\n";
8      std::cout << "Value pointed to by pnV is " << *pnV << "\n";
9  }
10
11 int main ()
12 {
13     int nV1;      // an integer variable
14     int *pnV1;    // pointer to an integer
15     int *pnV2;    // pointer to an integer
16
17     nV1 = 10;    // set the value
18     pnV1 = &nV1; // set the address
19     ShowValues (nV1, pnV1);
20
21     // manipulate the value
22     *pnV1 = 100; // changes value of nV1!
23     ShowValues (nV1, pnV1);
24
25     // pointer arithmetic
26     pnV2 = pnV1; // sets address of pnV2
27     *pnV2 = 400; // changes value of nV1!
28     ShowValues (nV1, pnV1);
29
30     return 0;
31 }

```

The program uses three variables that are declared in lines 13, 14 and 15. Two of these variables are pointer variables. The value of the `int` variable, nV1, is assigned in line 17. The memory address of this variable is obtained in line 18 and assigned to the pointer variable, pnV1. Then the function

`ShowValues` is called to display three lines of output associated with the `int` variable and the pointer variable associated with that `int` variable.

In line 3 we see how pointers can be used as function parameters. As a matter of style, we will use what is shown in line 3 rather than

```
void ShowValues (int nV, int* pnV)
```

just to drive home the point that the second parameter is a pointer to an `int`. In line 8, we display the `int` value at the memory address pointed to by `pnV`. When the usage (with a pointer variable) involves the asterisk symbol, `*`, the symbol is referred to as the **dereferencing** operator. The pointer variable must be dereferenced when the value stored at the memory location needs to be accessed. In other words, `pnV` refers to the memory address and `*pnV` refers to the value in memory pointed to by `pnV`.

Look at line 22. The value at the memory location pointed to by `pnV1` is set to 100. In the subsequent display of the values (see Fig. 8.2.1) we see that the value of `nV1` is also changed as a result of this statement. The memory address is expressed in hex (hexadecimal) and is likely to be different on different machines. In line 26, we see an assignment statement involving pointer variables. Note that

```
pnV2 = pnV1; // address assignment
```

is not the same as

```
*pnV2 = *pnV1; // value assignment
```

In line 27, the value at the memory location pointed to by `pnV2` is set to 400. In the subsequent display of the values (see Fig. 8.2.1) we see that the value `nV1` and consequently, `*pnV1`, is also changed as a result of this statement.

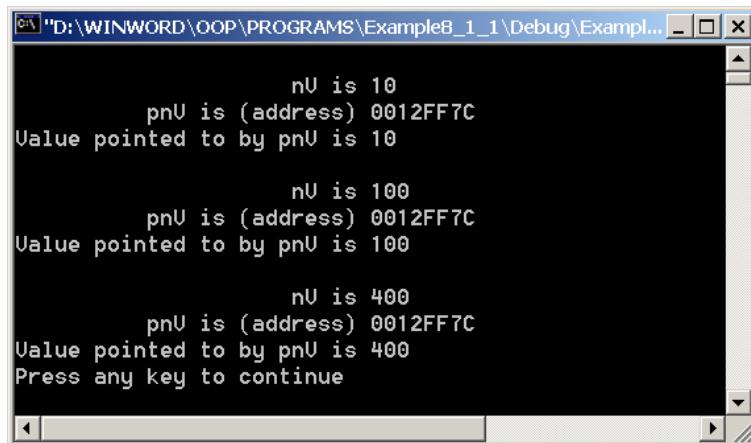


Fig. 8.2.1 Sample output created by Example Program 8.2.1

As we have seen in this simple example, the use of pointer variables can have unintended consequences if used incorrectly.

8.3 Dynamic Memory Allocation

Memory can be allocated and deallocated (or released) at run-time in C++ using two operators – `new` and `delete`.

`new` operator

In its simplest form, the `new` operator has the following syntax.

```
new type-name;
```

where `type-name` is a valid object. The `new` operator can be used to allocate objects and arrays of objects. This allocation takes place from a program memory area called the “heap” or “freestore.” We will see more about memory-related issues in the next section. When `new` is used to allocate a single object, it yields a pointer to that object; the resultant type is `type-name*`. When `new` is used to allocate a singly dimensioned array of objects, it yields a pointer to the first element of the array, and the resultant type is `type-name*`. Here is an example of declarations and usage.

```
float *pfX, fY;
pfX = new float; // allocates memory equal to one float
*pfX = 43.5f;    // use as any other float variable
fY = *pfX;
```

`delete` operator

It is a good practice to release the memory occupied by an object when that object is no longer needed. This is accomplished in C++ using the `delete` operator. In its simplest form, the `delete` operator has the following two forms.

```
delete pointer-object;
delete [] pointer-object;
```

where the first form is used for one object and the second form is for deallocating a number of objects. Here is an example illustrating both the `new` and the `delete` operators.

```
float *pfX, *pfVY;
pfX = new float;           // allocates memory equal to one float
pfVY = new float[40];     // allocates memory equal to 40 floats
...
delete pfX;               // releases memory back to freestore
delete [] pfVY;           // releases memory back to freestore
```

It is time now to look at a small but complete example.

Example Program 8.3.1 Using new and delete

We saw how classes were defined and used in Chapter 7. Recall the `CPoint` class from Section 7.2. To define and use this class we used statements such as

```
CPoint Point12;
```

The compiler generates the appropriate statements to obtain the memory required to store the instructions associated with the `CPoint` class and the memory locations to store the data associated with the object `Point12`. In this example, we will see another approach in obtaining the resources (memory) dynamically.

main.cpp

```

1  #include <iostream>
2  #include "..\Example7_2_1\point.h"
3
4  int main ()
5  {
6      CPoint *pPoint12;    // pointer to an object of class CPoint
7
8      // define an object of class CPoint
9      // memory is allocated
10     pPoint12 = new CPoint;    // default constructor is called
11
12     // set the x and y coordinate values
13     pPoint12->SetValues (1.2f, -17.65f);
14
15     // display the coordinate values
16     pPoint12->Display ("Point 12 ");
17
18     // release the memory
19     delete pPoint12;
20
21     return 0;
22 }
```

In line 6, `pPoint12` is declared as a pointer to a `CPoint` object. However, the memory to hold a `CPoint` object is allocated in line 10. The coordinate values are set in line 13. Note the usage of the member function `SetValues`. When an object is used to invoke the member function, the member selection operator `.` is used. When a pointer to an object is used, the member selection via pointer operator `->` (or arrow pointer) is used. C++ provides another way to write the statement in line 13 as

```
(*pPoint).SetValues (1.2f, -17.65f);
```

Finally, note line 19. The memory allocated in line 10 using `new` must be deallocated using the `delete` operator.

We will see more useful usages of dynamic memory allocations in later examples.

Functions with pointer parameters

We saw two different ways of passing parameters in Chapter 4 – call by value and call by reference. Now we will see examples of call by pointers (or call by reference via pointer arguments). Here is a simple example that illustrates the important concepts.

Example Program 8.3.2 Function calls and pointer arithmetic

In this example we will see two aspects dealing with pointers

- (a) call using pointer arguments, and
- (b) pointer arithmetic with vectors.

We will also see how to use the `const` qualifier to protect against inadvertent changes.

main.cpp

```

1  #include <iostream>
2
3  void CallViaPointers (const int *nA, int *nB)
4  {
5      // set B as two times A
6      *nB = 2*(*nA);
7  }
8
9  int main ()
10 {
11     // function call
12     int nA = 100;
13     int nB;           // value undefined
14
15     std::cout << "Before function call\n"
16         << "nA = " << nA << "    "
17         << "nB = " << nB << "\n";
18     CallViaPointers (&nA, &nB);
19     std::cout << "After function call\n"
20         << "nA = " << nA << "    "
21         << "nB = " << nB << "\n";
22     CallViaPointers (&nA, &nB);
23
24     // pointer arithmetic
25     int i;
26     double dVX[4] = {101.0, 102.0, 103.0, 104.0};
27     double *pdVX = dVX; // address of the first location in dVX
28
29     // display all the four values
30     std::cout << "\nValues in vector X\n";
31     for (i=0; i < 4; i++)
32     {
33         std::cout << "Location " << i << ":" "
34             << *(pdVX+i) << "\n";
35     }
36
37     return 0;

```

38 }

We define two `int` variables `nA` and `nB` in lines 12 and 13. Note that the value of `nB` is undefined. Hence the display generated by the program for `nB` is unpredictable. The function `CallViaPointers` is invoked in line 18. As we can see from the function definition in lines 3 through 7, the first parameter is a constant `int` pointer and the second parameter is a pointer to an `int`. In the function, the value of the second parameter is set as twice the value of the first parameter. Note the expression in line 6 and the manner in which the dereferencing operator is used. While the following statements will work

```
*nB = 2**nA;
*nB = 2* *nA;
```

they are more difficult to read and understand.

In line 26, a double precision vector `dVX` of length 4 is declared and initialized. In line 27, a double precision pointer is declared and the address is set as the starting location of the `dVX` vector. When the vector index is not defined, the first location or [0], is implied. Values in the vector can be accessed using pointers as in line 34. With this example, the four memory locations in the `dVX` vector can be accessed as `*(pdVX)`, `*(pdVX+1)`, `*(pdVX+2)` and `*(pdVX+3)`. In other words, by adding an integer to the memory address stored in a pointer, we are able to access other memory locations as long as the data type does not change and the resulting memory address is a legal address.

NULL pointer

C++ has a symbolic constant `NULL` that is used in a number of situations including with pointers. In the preceding examples, we have assumed that when the `new` operator is used to obtain memory space from the system, that the process works successfully. What if there is not enough memory space available and the system needs to communicate memory allocation failure to the program? Instead of returning a valid memory address, the `NULL` value is returned. In this specific usage, `NULL` has a zero value. The programmer must check to see if the memory allocation was unsuccessful and take an appropriate action (see Chapter 9 for C++ in-built exception handling capabilities).

Example Program 8.3.3 Vector data type

We can now finally show an example where the memory for a vector data type can be allocated and deallocated dynamically. The example is a modification of the second part of the previous example.

main.cpp

```
1 #include <iostream>
2 #include <stdlib.h> // contains definition of NULL and exit()
3
4 int main ()
5 {
6     float *fVX; // pointer to a float
7     // allocate memory locations for 4 float values
8     fVX = new float[4];
9     if (fVX == NULL)
10    {
```

```

11         std::cout << "Unable to allocate memory.\n";
12         exit (1);
13     }
14
15     int i;
16     // set the four values as 100, 101, 102, 103
17     for (i=0; i < 4; i++)
18     {
19         fVX[i] = 100.0f + static_cast<float>(i);
20     }
21
22     // display the values
23     for (i=0; i < 4; i++)
24     {
25         std::cout << "Location " << i << ":" << fVX[i]
26                     << "\n";
27     }
28
29     // release the memory locations allocated before
30     delete [] fVX;
31
32     return 0;
33 }
```

Instead of a static definition of the vector as

```
float fVX[4];
```

we have

```
float *fVX;
fVX = new float[4];
```

Note that the vector is dynamically allocated. We could have obtained, say from the user of the program, the size of the vector to be used in the program as follows.

```
int nSize;
GetInteractive ("What is the size of the vector? ", nSize);
float *fVX;
fVX = new float[nSize];
```

Once past the initial declaration and allocation, we have exactly the same usage for `fVX` in the program. Access to the elements of the vector is via the `[]` operator. In other words

```
*(fVX+i)
```

is equivalent to

```
fVX[i]
```

At the end of the program when `fVX` is no longer needed, memory is deallocated in line 30.

Tip: Vectors and pointers are intimately related. For example, if we have

```

float fVX[4];
float* pfVX = fVX;

```

then

```

pfVX = &(fVX[0]) is the same as pfVX = fVX
fVX[i] is the same as *(pfVX + i)
&(fVX[i]) is the same as (pfVX+i)
pfVX[i] is the same as fVX[i]

```

We will now take these ideas and develop a class to handle vector data type.

8.4 Case Study: A Poor Man's Vector Class

We finish this chapter's example with a case study involving the development of a vector class. As we have seen before, the C++ vector is statically allocated. In other words, the size of the vector must be declared when writing the program and cannot be changed. In the last section we saw how to dynamically allocate and deallocate storage space through the use of **new** and **delete** operators. In this section, we will show how we can start improving the features of dynamically controlled vector as follows.

- (a) First, we will declare and define a vector class, **CMyVector** used to store floating point numbers. The memory allocation and deallocation will take place within the class without the user having to explicitly allocate and deallocate. We will finally show where the class destructor can be used. This will take care of one of the problems with dynamic memory management – memory leaks, where memory is allocated but not deallocated.
- (b) Second, we will control the access to the vector elements two different ways. The first method will be via a public member variable – the pointer containing the memory address of the vector. The second method will be via a member function that will check to see if the vector index has a valid value. An assertion failure will take place if the index has an invalid value.
- (c) Third, we will show an example of a **reference** return type for a function.
- (d) Finally, we will illustrate how one can start building vector operations that are a part of the class's publicly available member functions.

Here is the declaration of the **CMyVector** class.

Example Program 8.4.1 Poor Man's Vector Class

CMyVector.h

```

1  #ifndef __RAJAN__MYVECTOR_H__
2  #define __RAJAN__MYVECTOR_H__
3
4  #include <string>
5

```

```

6   class CMyVector
7   {
8     public:
9       CMyVector () ;           // default constructor
10      CMyVector (int nRows); // constructor
11      ~CMyVector () ;       // destructor
12
13     // helper functions
14     int GetSize () const { return m_nRows; }
15     float At (int) const;
16     float& At (int);
17     void Display (const std::string&) const;
18
19     // vector operations
20     float DotProduct (const CMyVector&) const;
21
22   public:
23     float *m_pData;          // where the vector data are
24
25   private:
26     int m_nRows;             // # of rows
27 };
28 #endif

```

The default constructor is shown in line 9. The overloaded constructor in line 10 has a single parameter – the number of rows or elements in the matrix. As we will see below, the destructor shown in line 11, will be defined in this class. There are four helper functions. The `GetSize` function returns the number of rows in the vector. The `At` function used to access the elements of the vector, is overloaded. In line 15, we declare the version used to obtain a floating point value of the i^{th} element. In line 16, this version of the `At` function returns a `float` reference to the i^{th} element of the vector. The `Display` function is designed to display the elements of the vector. Finally, we show the genesis of a vector operations library in the form of the `DotProduct` function. We will leave the development of other useful vector-related functions as an exercise.

The attributes of the class, the pointer to contain the memory address and the number of rows in the vector, should be declared as private. However, we will declare the pointer, `m_pData` as a public variable to illustrate how the vector elements can (not should) be accessed.

Here is the implementation of the `CMyVector` class.

CMyVector.cpp

```

1  #include <cassert>
2  #include <cstdlib>
3  #include <iostream>
4  #include <iomanip>
5  #include "myvector.h"
6
7  // default constructor
8  CMyVector::CMyVector ()
9  {
10    m_nRows = 0;

```

```

11     m_pData = NULL;
12 }
13
14 // overloaded constructor
15 CMyVector::CMyVector (int nRows)
16 {
17     assert (nRows > 0);           // valid # of rows?
18
19     m_pData = new float[nRows];   // allocated memory
20     assert (m_pData != NULL);    // allocation successful?
21
22     m_nRows = nRows;            // store size of vector
23 }
24
25 // destructor
26 CMyVector::~CMyVector ()
27 {
28     if (m_nRows > 0)
29     {
30         delete [] m_pData;      // release memory
31         m_nRows = 0;
32     }
33 }
34
35 // vector access function via value
36 float CMyVector::At (int nIndex) const
37 {
38     // valid index?
39     assert (nIndex >= 0 && nIndex < m_nRows);
40     // return the value at location
41     return m_pData[nIndex];
42 }
43
44 // vector access function via reference
45 float& CMyVector::At (int nIndex)
46 {
47     // valid index?
48     assert (nIndex >= 0 && nIndex < m_nRows);
49     // return memory reference
50     return (m_pData[nIndex]);
51 }
52
53 // display vector values
54 void CMyVector::Display (const std::string& szHeader) const
55 {
56     // display specified header
57     std::cout << szHeader << "\n";
58     // show all values one value per line (using value based At fnc)
59     for (int i=0; i < m_nRows; i++)
60     {
61         std::cout << "[" << std::setw(2) << i << "]: "
62             << At(i) << "\n";
63     }
64 }
65
66 // dot product of vector with another
67 float CMyVector::DotProduct (const CMyVector& fV) const

```

```

68  {
69      int i;
70      float fDP = 0.0f;
71
72      // are the two vectors compatible?
73      assert (m_nRows == fV.m_nRows);
74      // now compute the dot product
75      for (i=0; i < m_nRows; i++)
76      {
77          fDP += At(i) * fV.At(i); // using value-based At function
78      }
79
80      return fDP;
81  }

```

The default constructor is used for initializing the member variables – number of rows to zero and the pointer variable to NULL; no memory is allocated. However, the overloaded constructor uses the function parameter to dynamically allocate the memory. Assertion failure occurs if the number of rows is less than or equal to zero or if the memory allocation fails. The destructor is defined in lines 26 through 33. Memory deallocation takes place here automatically in the sense that the destructor is called automatically when the object goes out of scope. Note that if the `delete` operator is used if the memory allocation did not take place then some compilers issue an error message. This is the reason for the check on the number of rows in line 28. We could have also written the check as

```
if (m_pData != NULL)
```

The value-based vector access function is defined in lines 36 through 42. A check is made in line 39 to see if the vector index has a valid value. An assertion failure occurs if the index value is invalid. Otherwise the value is returned. The reference-based vector access function is defined in lines 45 through 51. In this function, the memory reference is returned. What is the difference between lines 41 and 50 that appear to be identical? The use of the value-based function can take place only when the vector element is not being modified. In other words, if we had only the value-based function, the following statement

```
fVA.At(i) = 12.06f;
```

would not compile since a value cannot appear as a l-value (left of the assignment operator). On the other hand, pointers and references can appear as both l-value and r-value. When a reference to a value is returned from a function, the address (or reference) operator is **not** required.

Finally a note about the development of vector-related functions. The `DotProduct` function is defined in lines 67 through 81. Checks are made in the function to ensure that the operation is valid (see line 73). The vector elements are accessed in line 77 and the `At` function is used as a safety precaution. We could have rewritten the statement as

```
fDP += m_pData[i] * fV.m_pData[i];
```

Finally, we will see how to use the `CMyVector` class in a program. In this program we will dynamically allocate and populate two vectors and then compute their dot product.

main.cpp

```

1 #include <iostream>
2 #include "myvector.h"
3
4 int main ()
5 {
6     // dynamically allocated vectors
7     CMyVector fVX(3), fVY(3);
8
9     // populate the two vectors
10    int i;
11    for (i=0; i < 3; i++)
12    {
13        // populate using the public member variable
14        fVX.m_pData[i] = static_cast<float>(i+1);
15
16        // now populate using the reference-based At() function
17        fVY.At(i) = static_cast<float>((i+1)*(i+1));
18    }
19
20    // compute the dot product
21    float fDotP = fVX.DotProduct (fVY);
22
23    // display the result
24    std::cout << "Dot product of ... \n";
25    fVX.Display ("    Vector X ");
26    fVY.Display ("    Vector Y ");
27    std::cout << "    is equal to " << fDotP << "\n";
28
29    return 0;
30 }
```

The two vectors are declared in line 7. We show two different ways of populating the vectors in lines 14 and 17. The disadvantage with the usage on line 14 is that the vector index value is not checked for correctness whereas the check is carried out in the `At` function. The dot product is computed in line 21. One could have written the statement as

```
float fDotP = fVY.DotProduct (fVX);
```

The dot product result is displayed in lines 24 through 27 using the `Display` member function to display the contents of the two vectors.

There is a deficiency in the class definition. What if we declared a vector as follows.

```
CMyVector fVA;
```

The default constructor would be called setting the number of rows in the vector to zero. How do we then set the size of the vector later in the program? The solution to this problem is to define a member function, say `void SetRows (int nRows)` that would behave similar to the overloaded constructor. However, it would first check to see if memory for the vector has been allocated before. If memory has been allocated, it would then deallocate the memory and allocate new memory as specified by the function parameter.

What is one of the advantages of the reference return type? With the reference version of the At function, we can have cascading statements of the form

$$\text{fVX.At}(i) = \text{fVX.At}(i+1) = 3.1415926f;$$

As we mentioned in Chapter 4, functions can be made more useful if they can be converted to a function template. With this example, we can declare and manipulate only vectors containing float values. We will learn how to declare and define template classes in the next chapter.

Summary

In this chapter we saw more about memory management and especially, dynamic memory management. We learnt about pointers, pointer arithmetic, the `new` and `delete` operators, and the development of classes where memory allocation and deallocation can take place in a consistent and safe manner.

EXERCISES

Most of the problems below involve the development of one or more classes. In each case develop (a) plan to test the classes(s), and (b) implement the plan in a main program.

Appetizers

Problem 8.1

What is the output that is generated by the following statements?

```
int *pnA;
int nA = 5;
pnA = &nA;
std::cout << "nA is " << nA << " and *pnA is " << *pnA << '\n';
*pnA = 10;
std::cout << "nA is " << nA << " and *pnA is " << *pnA << '\n';
```

Problem 8.2

What is the output that is generated by the following statements? Is there an error in this program segment?

```
int *pnA, *pnB;
pnA = new int;
pnB = new int;
*pnA = 100;
*pnB = 110;
std::cout << *pnA << " " << *pnB << '\n';
pnA = pnB;
std::cout << *pnA << " " << *pnB << '\n';
*pnA = 300;
std::cout << *pnA << " " << *pnB << '\n';
delete pnA;
delete pnB;
```

Problem 8.3

Will the following statements compile? If not, correct the statements. What is the output that is generated?

```
int *pnVA;
int nVA[4] = {0, 1, 2, 3};
pnVA = nVA;
std::cout << *pnVA << '\n';
std::cout << pnVA[2] << '\n';
++pnVA;
std::cout << *pnVA << '\n';
std::cout << pnVA[2] << '\n';
```

Main Course

Problem 8.4

Extend the capabilities of the `CMyVector` class by adding a member function

```
void SetRows (int nRows);
```

that would dynamically either set the size or reset the size of a vector.

Problem 8.5

Extend the capabilities of the `CMyVector` class by adding other vector operations as public member functions.

Function Prototype	Remarks
<code>float MaxNorm () ;</code>	Computes the maximum absolute value. $\ \mathbf{x} \ _{\infty} = \max \{ x_i \}$
<code>float TwoNorm() ;</code>	Computes the length of the vector as $\ \mathbf{x} \ _2 = \sqrt{\sum_{i=1}^n x_i^2}$
<code>float MaxValue () ;</code>	Compute the largest value.
<code>float MinValue();</code>	Computes the smallest value.
<code>void UnitVector (const CMyVector& fVB, CMyVector& fVUnitVector);</code>	Computes the unit vector from current point to point <code>fVB</code> and stores the result in <code>fVUnitVector</code> .
<code>void CrossProduct (const CMyVector& fVB, CMyVector& fVR);</code>	Computes the cross product between the current vector and <code>fVB</code> and stores the result in <code>fVR</code> .

C++ Concepts

Problem 8.6

It is possible to have a pointer to a pointer. For example, the following

```
int **pMA;
```

defines a pointer `pMA` that points to a pointer. Consider the following code segment.

```
int* pVA[3]; // a vector of int pointers
```

```

int** pMA;      // pointer to a int pointer
int nV1[2] = {11, 12};
int nV2[2] = {21, 22};
int nV3[2] = {31, 32};

pVA[0] = nV1;    // stores the address of vector nV1
pVA[1] = nV2;    // stores the address of vector nV2
pVA[2] = nV3;    // stores the address of vector nV3

for (int i=0; i < 3; i++)
{
    pMA = &pVA[i]; // grab the address stored in pVA[i]
    for (int j=0; j < 2; j++)
    {
        std::cout << "[" << i << "," << j
                     << "] = " << pMA[0][j] << '\n';
        // notice how the dereferencing takes place above
        // using two indices
    }
}

```

There are two key statements in the above code.

`pMA = &pVA[i]; // grab the address stored in pVA[i]`

In the above statement the address stored in `pVA[i]` is assigned to `pMA`. The next important statement is the statement in which the `int` value is accessed via

`pMA[0][j]`

In general `pMA[i][j]` would imply the value stored at the memory location that is accessed as follows - Locate the memory address stored in `pMA[i]`, to that address add `j` and finally, get the value that is stored at that memory location!

Use this idea to create a class for storing two-dimensional arrays.

[Hint: See the book, Press et. al., *Numerical Recipes in C*, Cambridge Press.]

Demonstrate your implementation using the following class definition.

```

class CMyMatrix
{
public:
    CMyMatrix ();                                // default constructor
    CMyMatrix (int nRows, int nCols);           // constructor
    ~CMyMatrix ();                             // destructor

    // helper functions
    int GetRows();
    int GetColumns();
    float At (int, int) const;
    float& At (int, int);
    void Display (const std::string&) const;

public:

```

```
    float *m_pData;           // where the vector data are  
  
private:  
    int m_nRows;             // # of rows  
    int m_nColumns;          // # of columns  
};
```

Chapter 9

Classes: Objects 202

'I'm not smart, but I like to observe. Millions saw the apple fall, but Newton was the one who asked why.' Baruch, Bernard Mannes

'Knowledge is not achieved until shared.' Anon

'If a little knowledge is dangerous, where is the man who has so much as to be out of danger?'
Huxley, Thomas H

Ideas dealing with classes and associated objects were introduced in Chapter 7. In Chapter 8 we looked at pointers and resource (memory) allocation issues. In this chapter, we build more powerful class-related constructs using the ideas from both chapters. As building blocks, the ideas discussed in this chapter will form the basis for later chapters in which we will see even more object-oriented concepts implemented via C++.

Objectives

- To build on the earlier concepts associated with classes and objects.
- To understand the concepts associated with operator overloading, template classes, and dynamically managed arrays.
- To learn more about software engineering and the development of a matrix toolbox necessary for numerical analysis.

9.1 Operator Overloading

We learnt how to define our own data types using classes. C++ extends this capability by allowing operators normally associated with the standard data types to be also associated with user-defined classes. This is known as operator overloading. Let us look at an example to understand what this means. The `CPoint` class was developed in Chapter 7. It would be nice to extend the capabilities of the class to be able to carry out the following operations via these statements.

```
CPoint P1(10.2f, -4.9f), P2(-3.2f, 55.0f), P3;
P3 = P1 + P2;           // overloaded operator +
cout << P3 << "\n";    // overloaded operator <<
```

Unless the operators `+` and `<<` are overloaded, the above statements will not compile.

Before we learn the details of operator overloading, we need to understand (a) a special pointer called the `this` pointer, (b) what is meant by `friend` classes and functions, (c) how the `const` qualifier should be used, and (d) the reference operator `&`.

`this` pointer

There is a special pointer called `this` that provides every object access to itself. This pointer can be used to refer to both the member variables and member functions. Here is an example that uses `this` pointer in the (rewritten version of the) `Display` member function in the `CPoint` class.

```
// helper function
void CPoint::Display (const std::string& szBanner)
{
    // display the current coordinates
    std::cout << szBanner
        << "[X,Y] Coordinates = [ "
        << this->m_fXCoor << ","
        << (*this).m_fYCoor << " ].\n";
}
```

In this function, the use of `this` pointer is an unnecessary concoction since `m_fXCoor` is equivalent to `this->m_fXCoor`, etc. Note also that, `this->` is equivalent to `(*this).` as we obtain the value of the two coordinates two different ways. We will see a much better example of the use of `this` pointer with operator overloading.

`friend` classes and `friend` functions

The `private` member variables and functions as we saw in Chapter 7 cannot be accessed outside of the class. However, there is an exception. A `friend` of a class also has access to the `private` members. One can declare a function or an entire class to be a `friend` of a class. Here is an example of `CTriangle` class that is declared to be a friend of the `CPoint` class.

```
class CPoint
{
    friend class CTriangle;      // CTriangle class is a friend
public:
```

```

    // constructor
    CPoint ();
    // default
    ...
private:
    float m_fXCoor;
    float m_fYCoor;
};

```

The `CTriangle` class is declared to be a friend of the `CPoint` class in the `CPoint` class header file. The `friend` keyword is associated with neither public nor private qualifiers. Hence it should be declared as shown in the above example. With this declaration, we could have the following statements in a `CTriangle` class to access the two private member variables.

```

#include "point.h"
class CTriangle
{
public:
    // constructor
    CTriangle ();           // default
    ...
private:
    CPoint    m_Vertex[3];
};

void CTriangle::ComputeSide ()
{
    float fSide1 = Distance (m_Vertex[0].m_fXCoor, m_Vertex[0].m_fYCoor,
                            m_Vertex[1].m_fXCoor, m_Vertex[1].m_fYCoor);
    ...
}

```

A few things to note about the “friend” concept. If `CPoint` declares `CTriangle` as a friend, then `CPoint` does not automatically gain the friend status of `CTriangle`. This declaration must be explicitly made in the `CTriangle` class. In other words, a class must be explicitly identified as a friend class; the reciprocity idea does not apply here. Also, if `CTriangle` is a friend of `CPoint` class and `CPrism` is a friend of `CTriangle` class, then one cannot infer that `CPrism` is a friend of `CPoint` class. Proper use of the friend class concept can enhance the readability and performance of a program.

Finally, one can also declare and define friend function of a class. This is a function that is defined outside the class but has access to all the members of a class. We will see examples of this friend function with the overloaded `<<` and `>>` operators.

Proper use of `const`

Proper use of the `const` qualifier introduces a defensive programming mechanism within the source code. Let's review what we have learnt so far by looking at a few function prototypes.

```

int Add (int n1, int n2);           // no const used here
int Add (const int n1, const int n2); // alternate version

```

Both these declarations will work fine since the arguments are passed by value. However, the second

version adds an additional check so that if an attempt is made in the Add function to change the values of either n1 or n2, the compiler will issue an error message.

```
void DisplayMessage (std::string szMessage);           // no const used here
void DisplayMessage (std::string& szMessage);         // no const used here
void DisplayMessage (const std::string& szMessage);   // const used here
```

The three versions are quite different. The first case has pass-by-value argument. The copy constructor is called and a copy of the variable szMessage. If szMessage is modified in the function, the changes do not propagate back to the calling function. However, additional instructions are executed due to the creation of a local copy of the variable. The second version has pass-as-reference argument (no local copy of the variable is made). If the intent is to just display the message in the function, it is possible to inadvertently modify the message. However, this cannot take place in the third version due to the use of `const` qualifier.

Now let's look at the following case where another function is called from a function.

```
void DisplayMessage (std::string& szMessage);           // no const used here
...
int ComputeAbsSum (const std::string& szMessage,
                   int nV[], int nValues)
{
    int nSum = 0;
    if (nValues <= 0)
        DisplayMessage ("Invalid number of values");
    else
    {
        for (int i=0; i < nValues; i++)
            nSum += abs(nV[i]);
        DisplayMessage (szMessage);
    }
    return nSum;
}
```

This code will not compile since szMessage is declared as a `const` variable within the `ComputeAbsSum` function and cannot be passed to the `DisplayMessage` function where it is not a `const` string. In other words, if a parameter is declared as a `const` in one function then it must be declared as a `const` in all functions where it is used.

We can also use the `const` qualifier with return values from a function. For example, the following function prototype

```
const int HowMany ();
```

signifies a function that returns an integer. However, the returned value cannot be modified because of the `const` qualifier used before the function return type.

Finally, we can use the `const` qualifier with calling objects. We can have a member function designed so that it does not modify the value of the calling object.

```
class CPoint
{
public:
    void Display () const;
    ....
private:
    int m_nItems;
}
```

In this example, by placing `const` at the end of the `Display` function declaration, we tell the compiler not to allow the function to change the value of the calling object. The following function body is incorrect.

```
void CPoint::Display () const
{
    ++m_nItems;      // NOT correct
    std::cout << "Number of items is " << m_nItems << '\n';
}
```

The correct function body is as follows.

```
void CPoint::Display () const
{
    std::cout << "Number of items is " << m_nItems << '\n';
}
```

References

We saw the reference operator `&` used and discussed in Chapters 4 and 7. To make operator overloading work, we need to understand references to functions. Returning a reference to a function is similar to returning an alias to a variable. If an object returned by a function is to be an l-value then it must be returned by reference. Let `T` denote a class type. Consider the following function prototypes.

- (a) `T function ()`; where a value is returned and cannot be used as l-value. The returned value can be changed. The copy constructor is used in this case.
- (b) `const T function ()`; is the same as (a) but the returned value cannot be changed.
- (c) `T& function ()`; where a reference is returned can be used as l-value. The returned value can be changed. The copy constructor is not used in this case.
- (d) `const T& function ()`; where a const reference is returned cannot be used as l-value. The returned value cannot be changed. The copy constructor is not used in this case.

The differences will become clear as we look at a specific example.

Operator overloading

As we mentioned at the beginning of this section, C++ allows operator overloading with user-defined data types makes programs easier to read and maintain. The overloaded operators can be implemented either as global functions or as member functions of a class. The syntax is as follows.

operatorx

where `operator` is a C++ reserved keyword and `x` is the operator. Here are the definitions for the addition operator and the stream insertion operator.

```
operator+
```

```
operator<<
```

The general syntax, when used in the context of a function, is as follows.

```
returntype classname::operatorx ()
```

where `x` is the operator to be overloaded.

Operators that can be overloaded

The following operators can be overloaded.

Operator	Name	Type
,	Comma	Binary
!	Logical NOT	Unary
!=	Inequality	Binary
%	Modulus	Binary
%=	Modulus/assignment	Binary
&	Bitwise AND	Binary
&	Address-of	Unary
&&	Logical AND	Binary
&=	Bitwise AND/assignment	Binary
()	Function call	-
*	Multiplication	Binary
*	Pointer dereference	Unary
*=	Multiplication/assignment	Binary
+	Addition	Binary
+	Unary Plus	Unary
++	Increment ¹	Unary
+=	Addition/assignment	Binary

-	Subtraction	Binary
-	Unary negation	Unary
--	Decrement	Unary
--=	Subtraction/assignment	Binary
->	Member selection	Binary
->*	Pointer-to-member selection	Binary
/	Division	Binary
/=	Division/assignment	Binary
<	Less than	Binary
<<	Left shift	Binary
<<=	Left shift/assignment	Binary
<=	Less than or equal to	Binary
=	Assignment	Binary
==	Equality	Binary
>	Greater than	Binary
>=	Greater than or equal to	Binary
>>	Right shift	Binary
>>=	Right shift/assignment	Binary
[]	Array subscript	-
^	Exclusive OR	Binary
^=	Exclusive OR/assignment	Binary
	Bitwise inclusive OR	Binary
=	Bitwise inclusive OR/assignment	Binary
	Logical OR	Binary
~	One's complement	Unary
delete	delete	-
new	new	-

Operators that cannot be overloaded

Here is the list of operators that cannot be overloaded.

Operator	Name
.	Member selection
.*	Pointer-to-member selection
::	Scope resolution
? :	Conditional
#	Preprocessor symbol
##	Preprocessor symbol

There are two operators that do not explicitly require operator overloading. The first is the assignment operator **=** and the second is the address operator **&**. C++ automatically provides default behavior for both these operators.

We will now look at an example where we will overload operators to be used with the **CPoint** class including the **=** operator.

Example Program 9.1.1 Operator overloading

We will now develop the functionality within the now familiar **CPoint** class so that the following operations can be carried out using **CPoint** objects, P1, P2 and P3.

```
P1 = P2;                      // overloaded operator =
P3 = P1 + P2;                  // overloaded operator +
P3 = P1 - P2;                  // overloaded operator -
if (P1 == P2)                  // overloaded operator ==
if (P1 != P2)                  // overloaded operator !=
cout << P3 << "\n";          // overloaded operator <<
cin  >> P3 ;                  // overloaded operator >>
```

First we define the header file. Look at lines 31 through 35 to see how operators are overloaded as public functions.

Point.h

```
1  #ifndef __RAJAN_POINT_H__
2  #define __RAJAN_POINT_H__
3
4  #include <string>
5  #include <iostream>
6  using std::istream;
7  using std::ostream;
8
9  class CPoint
10 {
```

```

11     // friend overloaded operator functions
12     friend istream &operator>> (istream&, CPoint&);
13     friend ostream &operator<< (ostream&, const CPoint&);
14
15     public:
16         // constructor
17         CPoint ();           // default
18         CPoint (const CPoint&); // copy constructor
19         CPoint (float, float); // overloaded
20
21         // helper function
22         void Display (const std::string&) const;
23         // modifier function
24         void SetValues (float, float);
25         void SetValues (const CPoint&);
26         // accessor function
27         void GetValues (float&, float&) const;
28         void GetValues (CPoint&) const;
29
30         // overloaded operators
31         const CPoint& operator= (const CPoint&);
32         CPoint operator+ (const CPoint&) const;
33         CPoint operator- (const CPoint&) const;
34         bool operator!= (const CPoint&) const;
35         bool operator== (const CPoint&) const;
36
37     private:
38         float m_fXCoor;    // stores x-coordinate
39         float m_fYCoor;    // stores y-coordinate
40     };
41 #endif

```

Now the rest of the server code dealing with overloaded operator functions only is shown below.

point.cpp

```

1  // overloaded member operators
2  const CPoint& CPoint::operator= (const CPoint& PRight)
3  {
4      // copying itself?
5      if (&PRight != this)
6      {
7          // copy the coordinate values
8          m_fXCoor = PRight.m_fXCoor;
9          m_fYCoor = PRight.m_fYCoor;
10     }
11
12     return (*this);
13 }
14
15 CPoint CPoint::operator+ (const CPoint& PRight) const
16 {
17     CPoint PResult;
18
19     // add the coordinates

```

```

20         PResult.m_fXCoor = m_fXCoor + PRight.m_fXCoor;
21         PResult.m_fYCoor = m_fYCoor + PRight.m_fYCoor;
22
23     return PResult;
24 }
25
26 CPoint CPoint::operator- (const CPoint& PRight) const
27 {
28     CPoint PResult;
29
30     // subtract the coordinates
31     PResult.m_fXCoor = m_fXCoor - PRight.m_fXCoor;
32     PResult.m_fYCoor = m_fYCoor - PRight.m_fYCoor;
33
34     return PResult;
35 }
36
37 bool CPoint::operator== (const CPoint& PRight) const
38 {
39     // check whether both coordinates are equal
40     return (m_fXCoor == PRight.m_fXCoor &&
41             m_fYCoor == PRight.m_fYCoor);
42 }
43
44 bool CPoint::operator!= (const CPoint& PRight) const
45 {
46     // check whether one of the coordinates are unequal
47     return (m_fXCoor != PRight.m_fXCoor ||
48             m_fYCoor != PRight.m_fYCoor);
49 }
50
51 // overloaded operator friend functions
52 ostream &operator<< (ostream& ofs, const CPoint& Point)
53 {
54     // display the current coordinates
55     ofs << Point.m_fXCoor << "," << Point.m_fYCoor << ".\n";
56
57     return ofs;
58 }
59
60 istream &operator>> (istream& ifs, CPoint& Point)
61 {
62     // get the coordinate values
63     std::cout << "X Coordinate: ";
64     ifs >> Point.m_fXCoor;
65     std::cout << "Y Coordinate: ";
66     ifs >> Point.m_fYCoor;
67
68     return ifs;
69 }

```

Lines 2 and 12 show how the reference return type is used. By having the return type as a reference to a `CPoint` object, it is possible to have statements such as the following that involve multiple assignments with `CPoint` objects

```
P1 = P2 = P3;
```

In other words, the following prototype

```
void CPoint::operator= (const CPoint& PRight);
```

would permit only the following assignment

```
P1 = P2;
```

but not the two following statements

```
CPoint P1 = P2;
```

```
P1 = P2 = P3;
```

Line 5 is a defensive programming statement to avoid problems with statements like

```
P1 = P1;
```

and shows a nice use for the `this` pointer. The addition overloaded operator `+` has a `CPoint` return type to support statements such as

```
P1 = P2 + P3 + P4;
```

The function body clearly shows that a `CPoint` object is created in the function (line 17) to facilitate this addition. This is the downside to overloaded operators. In the case of a `CPoint` object not much additional resource is going to be used temporarily. However, with a larger object, creating temporary objects will be resource intensive.

The overloaded operators `<<` and `>>` are implemented as non-class friend functions. Once again, by having the return type as a reference type, it is possible to output multiple `CPoint` objects using a single statement, or read multiple points using a single statement.

And finally, here is an example program (client code) where the use of the overloaded operators is shown using the `CPoint` class.

main.cpp

```
1 #include "point.h"
2
3 int main ()
4 {
5     // declare and initialize two points P1 and P2
6     CPoint P1 (1.0f, 1.0f);
7     CPoint P2 (1.1f, 1.1f);
8
9     // use of copy constructor
10    CPoint P3 = P1;
11    P3.Display ("Point P3: ");
12
13    // use of overloaded + operator and copy constructor
```

```

14     CPoint P4 = P1 + P2;
15     P4.Display ("Point P4: ");
16
17     // overloaded != operator
18     if (P1 != P2)
19         std::cout << "Unequal coordinates.\n";
20     else
21         std::cout << "Equal coordinates.\n";
22
23     // overloaded >> operator
24     std::cin >> P1;
25     std::cin >> P2;
26
27     // overloaded = and - operators
28     P3 = P1 - P2;
29
30     // overloaded << operator
31     std::cout << "Point P3: " << P3 << "\n";
32
33     return 0;
34 }
```

The program statements are self-explanatory; however, the reader is encouraged to explore and confirm when temporary objects are created and when the copy constructor is called.

9.2 More about classes

In this section we will see a couple more class-associated features.

static members

There are times when a member variable defined in a class needs to be accessed by all the objects of that class. Such variables are called static variables. This variable acts like a global variable for the class objects. First, note that the static member variable is declared in the class declaration.

Here is an example to illustrate its usage. The static variable `m_nObjectsDefined` is designed to track how `CPoint` objects exist during the program execution.

```

class CPoint
{
public:
    // constructor
    CPoint ();           // default
    ...
private:
    float m_fXCoor;
    float m_fYCoor;
    static int m_nObjectsDefined; // static member variable
};
```

Second, note that the static variable is initialized outside of the class. Usually this is done at the beginning of the file containing the class definition. Third, in this example, it is used in the class constructor in a manner similar to any class variable.

```
int CPoint::m_nObjectsDefined = 0;
CPoint::CPoint ()
{
    ++m_nObjectsDefined;
}
.... // rest of the class definitions follow
```

In a similar manner, static member functions are member functions that do not access an object's data. In other words, a member function that does not access the member non-static variables, then it is possible to declare that member function as static. Continuing with the previous example, let us assume that we need a public function to obtain the current number of CPoint objects defined and that function is NumObjects(). The modified header file is as follows.

```
class CPoint
{
public:
    // constructor
    CPoint ();           // default
    static int NumObjects (); // static member function
    ...
private:
    float m_fXCoor;
    float m_fYCoor;
    static int m_nObjectsDefined;
};
```

Note that the **static** keyword is used to declare the member function but is not used in the member function definition itself. In other words, the static member function is defined as follows.

```
int CPoint::NumObjects ()
{
    return m_nObjectsDefined;
}
```

Note that the static function can access **m_nObjectsDefined** since that variable is a static variable.

forward class definitions

When one class is a friend of another, it is common for both classes to refer to the other class in the class definitions. This requires the use of forward declaration. Consider class A that uses class B as a friend class.

```
class B; // forward declaration
class A
{
    friend class B;
    ...
}
```

The forward declaration is nothing else but the keyword `class` followed by the class name and a terminating semicolon.

Example Program 9.2.1 Using Static Member Functions and Variables

In this example we will develop functions to add, subtract, multiply and divide two real number that are used as double precision variables. We will also track how many times each of these functions are called (as our own form of operations counter). These functionalities will be implemented in a class called `CMath`.

We first present the entire server code – header and function bodies for the `CMath` class.

`math.h`

```

1  ifndef __MATH_H__
2  define __MATH_H__
3
4  class CMath
5  {
6      public:
7          static double Add (double, double);
8          static double Subtract (double, double);
9          static double Multiply (double, double);
10         static double Divide (double, double);
11         static void DisplayCounters ();
12
13     private:
14         static int nAddCounter;
15         static int nSubtractCounter;
16         static int nMultiplyCounter;
17         static int nDivideCounter;
18     };
19
20 #endif

```

While this class shows all member variables and functions to be `static`, we can design classes where only one or more functions and variables are declared to be `static` in a similar fashion.

`math.cpp`

```

1  #include <iostream>
2  #include "math.h"
3
4  int CMath::nAddCounter = 0;
5  int CMath::nSubtractCounter = 0;
6  int CMath::nMultiplyCounter = 0;
7  int CMath::nDivideCounter = 0;
8
9  double CMath::Add (double d1, double d2)
10 {
11     ++nAddCounter;

```

```

12     return (d1 + d2);
13 }
14
15 double CMath::Subtract (double d1, double d2)
16 {
17     ++nSubtractCounter;
18     return (d1 - d2);
19 }
20
21 double CMath::Multiply (double d1, double d2)
22 {
23     ++nMultiplyCounter;
24     return (d1 * d2);
25 }
26
27 double CMath::Divide (double d1, double d2)
28 {
29     ++nDivideCounter;
30     return (d1 / d2);
31 }
32
33 void CMath::DisplayCounters ()
34 {
35     std::cout << "\nOperations Count\n"
36             << "Addition      " << nAddCounter      << " time(s)\n"
37             << "Subtraction    " << nSubtractCounter << " time(s)\n"
38             << "Multiplication " << nMultiplyCounter << " time(s)\n"
39             << "Division       " << nDivideCounter   << " time(s)\n";
40 }

```

Statements 4-7 initialize the values of the static variables. Note that the `static` keyword is not used with the function definition. This `static` keyword is used only in the header file with the function declarations. Defensive programming would dictate that we check for math errors; for example, we should check to see if `d2` is zero before dividing on line 30.

Finally we present the client code.

main.cpp

```

1 #include <iostream>
2 #include "math.h"
3
4 int main ()
5 {
6     std::cout << "1.0 + 2.0 = " << CMath::Add (1.0, 2.0)      << '\n';
7     std::cout << "1.0 - 2.0 = " << CMath::Subtract (1.0, 2.0) << '\n';
8     std::cout << "1.0 * 2.0 = " << CMath::Multiply (1.0, 2.0) << '\n';
9     std::cout << "1.0 / 2.0 = " << CMath::Divide (1.0, 2.0)   << '\n';
10
11     CMath::DisplayCounters ();
12
13     return 0;
14 }

```

It should come as a surprise to us that there is no explicit variable (object) associated with the CMath class in the above code. Note how the scope resolution operator `::` is used to invoke the static member functions that are treated differently. A transient object is created automatically and destroyed.

9.3 Template Classes

In a fashion similar to template functions that we saw in Chapter 4, template classes can be defined. Let us go back to the CPoint class. Let us assume that the point coordinates can be of any of the following data type – integer, float or double. We can define a template CPoint class as follows (contained in `point.h`).

```
template <class T>
class CPoint
{
    // friend overloaded operator functions
    friend istream &operator>> (istream&, CPoint&);
    friend ostream &operator<< (ostream&, const CPoint&);

public:
    // constructor
    CPoint ();           // default
    CPoint (const CPoint&); // copy constructor
    CPoint (T, T);       // overloaded

    // helper function
    void Display (const std::string&) const;
    // modifier function
    void SetValues (T, T);
    void SetValues (const CPoint&);
    // accessor function
    void GetValues (T&, T&) const;
    void GetValues (CPoint&) const;

    // overloaded operators
    const CPoint &operator= (const CPoint&);
    CPoint operator+ (const CPoint&) const;
    CPoint operator- (const CPoint&) const;
    bool operator!= (const CPoint&) const;
    bool operator== (const CPoint&) const;

private:
    T m_XCoor;   // stores x-coordinate
    T m_YCoor;   // stores y-coordinate
};
```

There are just two differences when this version is compared to the float-based CPoint version shown in the last section. The first line reads

```
template <class T>
```

signifying that we have one template parameter for the class that is identified as T. Second, wherever we had specified `float` as the data type, we now specify the corresponding data type as T. The template function body must follow a slightly different syntax. For example, the `SetValues` function would be defined as follows.

```
template <class T>
void CPoint<T>::SetValues (T fX, T fY)
{
...
}
```

And the copy constructor would be defined as follows.

```
template <class T>
CPoint<T>::CPoint (const CPoint<T>& P)
{
...
}
```

In addition to the first line containing the template keyword, the `<T>` needs to be appended to the class name before the function name and everywhere else it is used (e.g. as a function parameter).

To use the `CPoint` template class in a program we would do the following.

```
#include "point.h"
...
CPoint<int> nPA, nPB; // integer-valued points
CPoint<float> fPA, fPB; // float-valued points
```

instead of the usual

```
CPoint fPA, fPB; // float-valued points
```

Function templates are not restricted to one template parameter. For example, if there are two parameters to be used with the `CPoint` class, then the class definition would be as follows.

```
template <class T1, class T2>
class CPoint
{
    // friend overloaded operator functions
    friend istream &operator>> (istream&, CPoint&);
...
};
```

A typical function body would then be defined as follows.

```
template <class T1, class T2>
void CPoint<T1,T2>::SetValues (T1 fX, T2 fY)
{
...
}
```

```

}

template <class T1, class T2>
CPoint<T1,T2>::CPoint (const CPoint<T1,T2>& P)
{
}

```

C++ also allows the keyword `typename` to be used instead of the `class` keyword. For example.

```

template <typename T1, typename T2>
CPoint<T1,T2>::CPoint (const CPoint<T1,T2>& P)
{
}

```

We will see examples of template classes in the next section.

9.4 Arrays

One of the most important objects or data structures that we will deal with is the area of numerical analysis is an array – vectors and two-dimensional matrices. Almost always, it can be assumed that the algorithm is able to ascertain *a priori*, the size (number of rows and/or columns) of the array. With this scenario, arrays provide the most convenient data structure to store and manipulate engineering data.

We will use the template approach in defining vector and matrix classes.

9.4.1 Vector Container Class¹

In Section 8.3, we defined and used the `CMyVector` class. In this section, we will use a similar but improved version of that class called `CVector`.

We will now define the attributes and behavior of arrays starting with the vector class. Recall that a vector either is a row vector or a column vector. The `CVector` class is general enough to store any data type. It has the following properties.

- (a) Both row and column vectors will be stored as a vector with n storage locations.
- (b) The indexing will start at 1. In other words, the indexing will be between 1 and n (both inclusive). The `()` operator will be overloaded and will be used to access the elements of the vector, e.g. `nV(j)` will point to the j^{th} element of the vector.
- (c) An assertion failure will occur if the vector index does not have a legal value. This check will be carried out only for the DEBUG version of a program.
- (d) Member functions will be provided to dynamically allocate as well as change the size of the vector.

¹ Strictly speaking, containers have a number of properties such as iterators, overloaded operators etc. that we do not support with the `CVector` and `CMatrix` classes. However, see Section 10.10.

- (e) The **=** operator will be overloaded.

Our template-based vector class is defined below and is followed by a sample program that uses the **CVector** class. Some of the member functions are discussed below.

Function Name	Remarks
SetSize	The initial size of the vector is determined by the constructor used. The default constructor sets the size as zero. This public function can be used to set or reset the size. If the size is reset, the original contents are destroyed.
Release	This protected ² function is used to release the memory allocated to store the elements of the vector.
GetSize	This public function returns the current size of the vector.
Set	This public function is used to set the specified value for all the elements of the vector.
ErrorHandler	This protected function is used to display the error message during the execution of the debug version of the program.

We first present the source code for the **CVector** template class that is contained in a header file.

vectortemplate.h

```

1  #ifndef __RAJAN_VECTORTEMPLATE_H__
2  #define __RAJAN_VECTORTEMPLATE_H__
3
4  #include <iostream>
5  #include <cassert>
6
7  // defines the vector template class

```

² We will see **protected** functions in greater detail in Chapter 13. However, keep in mind that access to member functions and variables is restricted as follows.

	public	protected	private
Members of the same class or friend classes	Yes	Yes	Yes
Members of the derived class	Yes	Yes	No
Others	Yes	No	No

```

8   template <class T>
9   class CVector
10  {
11      protected:
12          int nRows;           // number of rows in the vector
13          T *cells;          // address where the vector of type T is stored
14          void ErrorHandler (int,int nR=0) const;
15                      // handles error conditions
16          void Release (); // similar to destructor
17
18      public:
19          CVector ();           // default constructor
20          CVector (int);        // constructor
21          CVector (const CVector<T>&); // copy constructor
22          ~CVector ();         // destructor
23          void SetSize (int);  // sets the size of the vector
24                      // used with the default constructor
25
26          // vector size manipulation functions
27          // including memory allocations and deallocations
28          int GetSize () const; // gets the current size of the vector
29
30          // vector manipulations (mutator)
31          void Set (T);        // sets the value of all
32                      // elements of a vector
33
34          // overloaded operators
35          T& operator() (int); // row access
36          const T& operator() (int) const; // row access
37          CVector<T>& operator= (const CVector<T>&); // overloaded =
38      };
39
40 // ===== definitions =====
41 template <class T>
42 CVector<T>::CVector ()           // constructor
43 {
44     cells = NULL;
45     nRows = 0;
46 }
47
48 template <class T>
49 CVector<T>::CVector (int n)       // constructor
50 {
51     cells = NULL;
52     nRows = 0;
53     SetSize (n);
54 }
55
56
57 template <class T>
58 CVector<T>::CVector (const CVector<T>& A) // copy constructor
59 {
60     cells = NULL;
61     nRows = A.GetSize();
62     SetSize (nRows);
63     for (i=1; i <= nRows; i++)
64         cells[i] = A.cells[i];
65 }
```

```

66
67 template <class T>
68 void CVector<T>::SetSize (int nR) // sets the size as nR rows
69 {
70     // check whether NR is legal
71     if (nR <= 0) ErrorHandler (3);
72     Release ();
73     cells = new T [nR + 1];
74     if (cells == NULL) ErrorHandler (1);
75     nRows = nR;
76 }
77
78 template <class T>
79 CVector<T>::~CVector () // destructor
80 {
81     // deallocate storage
82     Release ();
83 }
84
85 template <class T>
86 void CVector<T>::Release () // similar to destructor
87 {
88     // deallocate storage
89     if (cells != NULL)
90     {
91         delete [] cells;
92         cells = NULL;
93         nRows = 0;
94     }
95 }
96
97 // ===== member functions =====
98 template <class T>
99 void CVector<T>::Set (T dV)
100    // sets the contents of the vector to dV
101 {
102     for (int i=1; i <= nRows; i++)
103         cells[i] = dV;
104 }
105
106 template <class T>
107 int CVector<T>::GetSize () const
108 {
109     return nRows;
110 }
111
112 // ===== Overloaded Operators =====
113 // overload () for use as bounds-checking
114 #ifdef _DEBUG
115 template <class T>
116 T& CVector<T>::operator() (int nR) // T& is reference
117 {
118     // row-column reference in bounds?
119     if (nR <= 0 || nR > nRows)
120     {
121         ErrorHandler (2,nR);
122         return cells[1];

```

```

123      }
124      else
125          return cells[nR];
126  }
127 #else
128 template <class T>
129 inline T& CVector<T>::operator() (int nR)
130 {
131     return cells[nR];
132 }
133 #endif
134
135 #ifdef _DEBUG
136 template <class T>
137 const T& CVector<T>::operator() (int nR) const
138 {
139     // row-column reference in bounds?
140     if (nR <= 0 || nR > nRows) {
141         ErrorHandler (2,nR);
142         return cells[1];
143     }
144     else
145         return cells[nR];
146 }
147 #else
148 template <class T>
149 inline const T& CVector<T>::operator() (int nR) const
150 {
151     return cells[nR];
152 }
153 #endif
154
155 // overload = for vector equality
156 template <class T>
157 CVector<T>& CVector<T>::operator= (const CVector& matarg)
158 {
159     // check whether vector is assigned to itself
160     if (this != &matarg)
161     {
162         // compatible vectors?
163         if (nRows != matarg.nRows)
164         {
165             ErrorHandler (4);
166             return *this;
167         }
168         // now copy
169         for (int i=1; i <= matarg.nRows; i++)
170             cells[i] = matarg.cells[i];
171     }
172
173     return *this;
174 }
175
176 // ===== Error Handler =====
177 template <class T>
178 void CVector<T>::ErrorHandler (int nErrorCode, int nR) const
179 {

```

```

180     switch (nErrorCode)
181     {
182         case 1:
183             std::cerr << "Vector::Memory allocation failure.\n";
184             break;
185         case 2:
186             std::cerr << "Vector::Row index is out of bounds.\n";
187             break;
188         case 3:
189             std::cerr << "Vector::Invalid number of rows or columns.\n";
190             break;
191         case 4:
192             std::cerr << "Vector::Ctor. Incompatible vectors.\n";
193             break;
194     }
195     exit (0);
196 }
```

Adding additional functionalities such as overloading the `<<` and `>>` operators, are left as an exercise.

Example Program 9.4.1 Using the CVector class

In this example, we write a program to define two vectors A and B, add the contents of the two vectors, store the result in a third vector, C, and display the contents of C. We will store and manipulate these vectors using the CVector class. This example can then be generalized for other types of applications.

main.cpp

```

1 #include <iostream>
2 #include <string>
3 #include <iomanip>
4 #include "..\library\vectortemplate.h"
5
6 void Display (const std::string& szHeader, const CVector<float>& fV)
7 {
8     std::cout << szHeader << "\n";
9     for (int i=1; i <= fV.GetSize(); i++)
10    {
11        std::cout << "(" << std::setw(2) << i << "): "
12            << fV(i) << "\n";
13    }
14 }
15
16 int AddVectors (const CVector<float>& fV1, const CVector<float>& fV2,
17                  CVector<float>& fV3)
18 {
19     if ((fV1.GetSize() != fV2.GetSize()) ||
20         (fV1.GetSize() != fV3.GetSize()))
21         return 1;
22
23     for (int i=1; i <= fV1.GetSize(); i++)
24         fV3(i) = fV1(i) + fV2(i);
25 }
```

```

26         return 0;
27     }
28
29     int main ()
30     {
31         CVector<float> fVA(3), fVB(3); // vectors with 3 rows/columns
32         fVA.Set (0.0f);           // clear the vectors to zero
33         fVB.Set (0.0f);           // clear the vectors to zero
34
35         CVector<float> fVC(3); // define a vector with 3 rows/columns
36
37         // set the values for vectors A and B
38         for (int i=1; i <= 3; i++)
39         {
40             fVA(i) = static_cast<float>(i);
41             fVB(i) = static_cast<float>(2*i);
42         }
43         Display ("Vector A", fVA);
44         Display ("Vector B", fVB);
45
46         // add the vectors with the result in vector C
47         AddVectors (fVA, fVB, fVC);
48         Display ("Vector C", fVC);
49
50         // the following statement will lead to
51         // an assertion failure during execution (debug version)
52         fVC(4) = 10.0f;
53
54     return 0;
55 }
```

The client code enhances the capabilities by adding two non-member functions – `Display` and `AddVectors`. The main program defines the values in two vectors A and B and then stores the sum of those two vectors in another vector, C. Note that we could have asked the user to specify the size of the A and B vectors at run time and then dynamically allocated memory for the three vectors as follows.

```

int nVecSize;
std::cout << "What is the size of the vectors?";
std::cin >> nVecSize;
CVector<float> fVA(nVecSize), fVB(nVecSize), fVC(nVecSize);
```

Tip: Note how line 52 fails because of the illegal (vector) index value. This is the most common programming error with the usage of vectors and matrices. Placing a breakpoint in the `ErrorHandler` member function in the `CVector` class helps in detecting the offending statement! The standard usage with C++ vector and `std::vector` fails to detect such errors.

```

float fVX[10];
vector<float> fVXX(10);
...
fVX[10] = 23.5; // illegal access
fVXX[10] = 23.5; // illegal access
```

9.4.2 Matrix Container Class

In a manner similar to the `CVector` class, we will now define a `CMatrix` class. The `CMatrix` class is general enough to store any data type. It has the following properties.

- (a) The matrix has n rows and m column.
- (b) The indexing will start at 1. In other words, the indexing will be between 1 and n (both inclusive) for the row number and 1 and m (both inclusive) for the column number. The `(` operator will be overloaded and will be used to access the elements of the matrix, e.g. `nMC(n, 3)`.
- (c) An assertion failure will occur if either the row or the column index does not have a legal value. This check will be carried out only for the DEBUG version of a program.
- (d) Member functions will be provided to dynamically allocate and change the size of the matrix.
- (e) The `=` operator will be overloaded.

To understand the manner in which the `CMatrix` class is implemented requires us to understand what is meant by “pointer to a pointer”. For example, the following

```
int **pMA;
```

defines an `int` pointer `pMA` that points to an `int` pointer. Recall that a pointer variable is designed to hold a memory address of the variable that it is pointing to. Consider the following code segment.

```
int* pVA[3]; // a vector of int pointers
int** pMA; // pointer to an int pointer
int nV1[2] = {11, 12};
int nV2[2] = {21, 22};
int nV3[2] = {31, 32};

pVA[0] = nV1; // stores the address of vector nV1
pVA[1] = nV2; // stores the address of vector nV2
pVA[2] = nV3; // stores the address of vector nV3

for (int i=0; i < 3; i++)
{
    pMA = &pVA[i]; // grab the address stored in pVA[i]
    for (int j=0; j < 2; j++)
    {
        std::cout << "[" << i << "," << j
            << "] = " << pMA[0][j] << '\n';
        // notice how the dereferencing takes place above
        // using two indices
    }
}
```

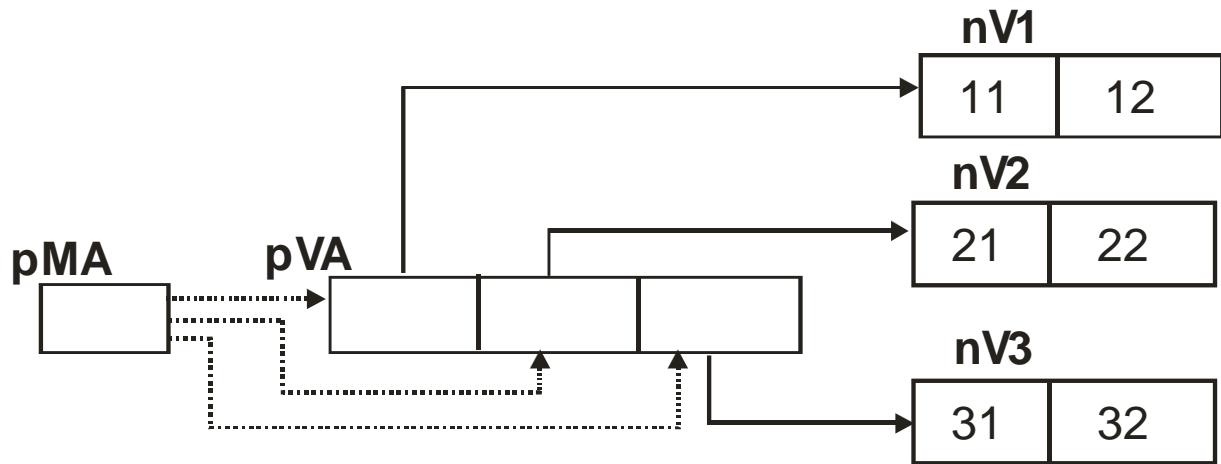


Fig. 9.4.2.1 Memory map

The memory map is shown in Fig. 9.4.2.1. Arrows emanate from **pMA** and **pVA** since the pointer variables point to memory locations. There are two key statements in the above code.

```
pMA = &pVA[i]; // grab the address stored in pVA[i]
```

In the above statement the address stored in **pVA**[*i*] is assigned to **pMA**. These refer to the three dotted lines in Fig. 9.4.2.1. The next important statement is the statement in which the **int** value is accessed via

```
pMA[0][j]
```

In general **pMA**[*i*][*j*] would imply the value stored at the memory location that is accessed as follows. Locate the memory address stored in **pMA**[*i*], to that address add *j* and finally, get the value that is stored at that memory location!

We can improve on this implementation by combining what the variables **pVA** and **pMA** do (that is to store memory addresses). A refined memory map is shown in Fig. 9.4.2.2. If we store the starting address of each row in the matrix in **pMA**, then **&pMA**[*i*] would have the starting address of row(*i*-1). Since values in each row are stored in adjacent memory locations, **pMA**[*i*][*j*] would point to the value stored at row *i* and column *j*!

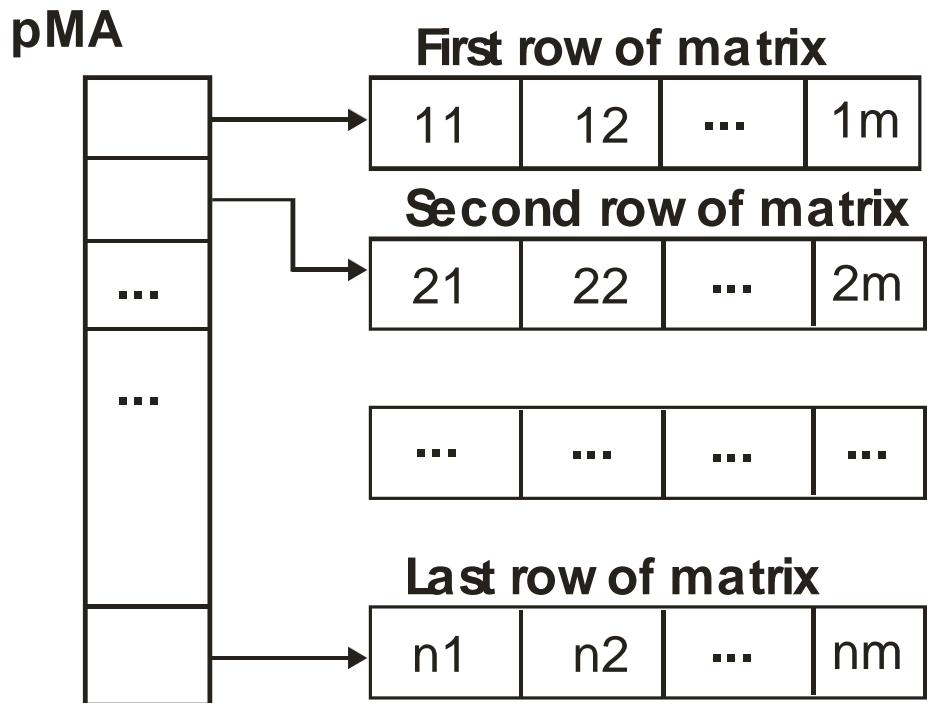


Fig. 9.4.2.2 Refined memory map (n: rows, m: columns)

The actual implementation in the `CMatrix` class is slightly different. `nR` is the number of rows and `nC` the number of columns in the matrix. The first two statements of interest are as follows.

```
T** cells;
cells = new T *[nR + 1];
```

The first statement declares a pointer to a pointer of type `T` (e.g. `int`). The second statement allocates memory for $(nR+1)$ locations to store the starting address of each row. The plus 1 is to facilitate counting from 1 rather than zero; hence we have one wasted memory space! Next we compute the total number of memory locations needed to store the entire matrix.

```
int size = nR*nC + 1;
```

Once again the plus 1 is to facilitate our indexing scheme for rows and columns that starts at 1. Now we allocate the memory space for the entire matrix.

```
cells[0] = new T[size];
```

Note `cells[0]` contains the starting memory address for the entire matrix or in other words, the first row. Since our indexing scheme starts at 1 not zero, we need to adjust the addressing scheme starting with

```
cells[1] = cells[0];
```

so that when we use `cells[1]` it does point to the starting address of the first row. This is followed by setting the starting address for each row as follows.

```
for (i=2; i <= nR; i++)
    cells[i] = cells[i-1] + nC;
```

The above statement simply computes the starting address of rows 2 through the last row, as the starting address of the previous row plus the number of columns in that row (or the matrix).

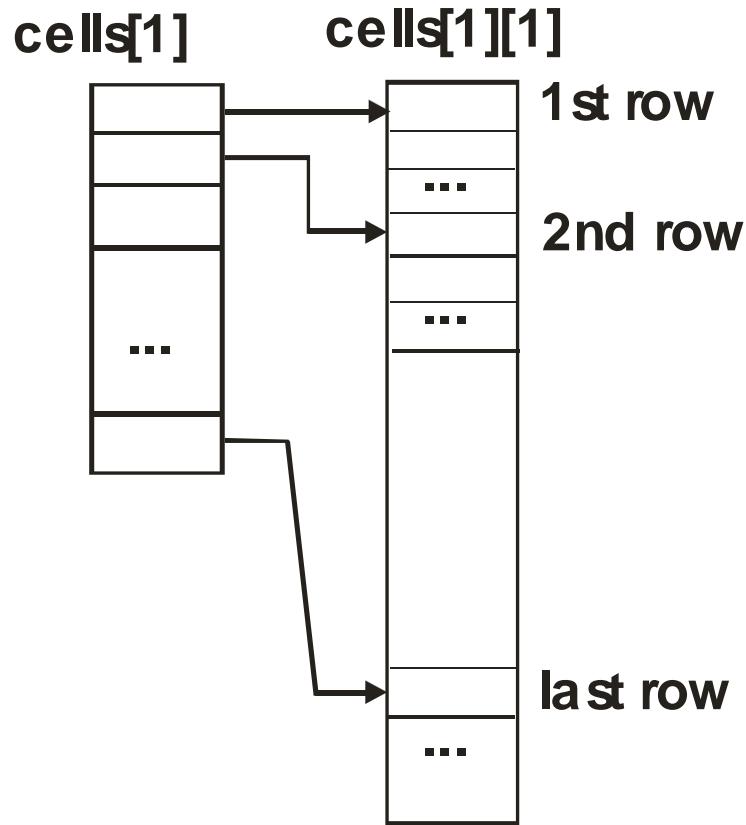


Fig. 9.4.2.3 Memory mapping implementation diagram for `CMatrix` class (wasted space not shown)

The above implementation is for a row-oriented matrix. However, a similar implementation can be devised for a column-oriented matrix.

Our template-based matrix class is defined below and is followed by a sample program that uses the `CMatrix` class. For brevity sake, the entire source code is not shown here, just the class definition.

```
Class definition for Matrix Template Class
// defines the Matrix class
template <class T>
class CMatrix {
protected:
    T      **cells;      // address where the matrix is stored
    int    nRows;        // number of rows in the matrix
```

```

        int nColumns; // number of columns in the matrix
        void ErrorHandler (int,int nR=0, int nC=0);
                                // handles error conditions
        void Release (); // similar to destructor

    public:
        CMATRIX (int, int); // constructor
        CMATRIX (); // default constructor
        CMATRIX (const CMATRIX<T>&); // copy constructor
        ~CMATRIX (); // destructor

        // matrix size manipulation functions
        // including memory allocations and deallocations
        void SetSize (int, int); // sets the size of the matrix
        int GetRows () const; // gets the current number of rows
        int GetColumns () const; // gets the current number of columns

        // matrix manipulations (mutator)
        void Set (T); // sets the value of all elements
                        // of a matrix
        T& operator() (int, int); // row-col access
        const T& operator() (int, int) const; // row-col access
        T& operator= (const CMATRIX&); // overloaded = operator
    };
}

```

Example Program 9.4.2 Using the CMATRIX class

In this example, we write a program to define two matrices A and B, add the contents of the two matrices, store the result in a third matrix, C, and display the contents of C. We will store and manipulate these matrices using the CMATRIX class.

main.cpp

```

1 #include <iostream>
2 #include <string>
3 #include <iomanip>
4 #include "..\library\matrixtemplate.h"
5
6 void Display (const std::string& szHeader, const CMATRIX<float>& fM)
7 {
8     std::cout << szHeader << "\n";
9     for (int i=1; i <= fM.GetRows(); i++)
10    {
11        for (int j=1; j <= fM.GetColumns(); j++)
12        {
13            std::cout << "(" << std::setw(2) << i << ", "
14                            << std::setw(2) << j
15                            << "): " << fM(i,j) << "\n";
16        }
17    }
18 }
19
20 int AddMatrices (const CMATRIX<float>& fM1, const CMATRIX<float>& fM2,
21                  CMATRIX<float>& fM3)
22 {

```

```

23     if ((fM1.GetRows() != fM2.GetRows()) ||
24         (fM1.GetRows() != fM3.GetRows()))
25     {
26     return 1;
27     if ((fM1.GetColumns() != fM2.GetColumns()) ||
28         (fM1.GetColumns() != fM3.GetColumns()))
29     {
30     return 1;
31     for (int i=1; i <= fM1.GetRows(); i++)
32     {
33     for (int j=1; j <= fM1.GetColumns(); j++)
34     {
35     fM3(i,j) = fM1(i,j) + fM2(i,j);
36     }
37     }
38     return 0;
39   }
40
41 int main ()
42 {
43   CMatrix<float> fMA(3,2), fMB(3,2); // 3 rows and 2columns
44   fMA.Set (0.0f);                      // clear the matrix to zero
45   fMB.Set (0.0f);                      // clear the matrix to zero
46
47   CMatrix<float> fMC(3,2); // define the matrix to hold the result
48
49   // set the values for matrices A and B
50   for (int i=1; i <= 3; i++)
51   {
52     for (int j=1; j <= 2; j++)
53     {
54       fMA(i,j) = static_cast<float>(i+j);
55       fMB(i,j) = static_cast<float>(i+2*j);
56     }
57   }
58
59   Display ("Matrix A", fMA);
60   Display ("Matrix B", fMB);
61
62   // add the matrices with the result in matrix C
63   AddMatrices (fMA, fMB, fMC);
64   Display ("Matrix C", fMC);
65
66   // the following statement will lead to
67   // an assertion failure during execution (debug version)
68   fMC(4,1) = 10.0f;
69
70   return 0;
71 }
```

The client code enhances the capabilities by adding two non-member functions – `Display` and `AddMatrices`. The main program defines the values in two matrices A and B and then stores the sum of those two matrices in another matrix, C. Note that we could have asked the user to specify the size of these matrices at run time and then dynamically allocated memory for the three matrices as follows.

```

int nR, nC;
std::cout << "How many rows and columns?";
std::cin >> nR >> nC;
CMATRIX<float> fMA(nR, nC), fMB(nR, nC), fMC(nR, nC);

```

Tip: The overloading of the operators () makes it possible to detect indexing errors with the CMATRIX class just as we saw the CVECTOR class handle this problem.

9.5 Exception Handling

We first saw exception handling in Chapter 4. In this section we will see how C++ provides a hierarchy of classes to handle exceptions. Two classes are immediately derived from the class exception: `logic_error` and `runtime_error`. Both these classes are defined in the header file `stdexcept`.

Several classes are derived from the class `logic_error`. Class `invalid_argument` and class `out_of_range` are provided to trap illegal arguments in a function call and string subscripts out of range error respectively. `domain_error` If memory allocation through the use of `new` operator cannot take place, the `bad_alloc` exception error is thrown.

The class `runtime_error` is designed to deal with errors that occur when a program is executing. Such errors include underflow and overflow during arithmetic computations through `underflow_error` and `overflow_error` classes. In addition, indexing errors in accessing elements of an array can be detected using `range_error`.

In the following example we look at using these exception handling classes.

Example Program 9.5.1

We look at detecting three errors two of which are logical errors and one runtime error.

```

1 #include <iostream>
2 #include <stdexcept>
3 #include <string>
4 #include <vector>
5 #include <cmath>
6
7 int main ()
8 {
9     // Logical Error 1: Bad Memory Allocation
10    double *pBigArray;
11    int nSize = 1024;
12    bool bError = false;
13    do
14    {
15        try
16        {
17            pBigArray = new double [nSize];
18        }
19        catch (std::bad_alloc)
20        {
21            std::cout << "Cannot allocate : " << nSize << " DP words\n";
22            bError = true;
23        }

```

```

24         if (!bError)
25     {
26         std::cout << "Allocated : " << nSize << " DP words\n";
27         delete [] pBigArray;
28         nSize *= 2;
29     }
30 } while (!bError);
31
32 // Logical Error 2: string extraction error
33 std::string szFileName = "thisANDthat.txt";
34 std::string szExtension = "?";
35 try
36 {
37     size_t nLoc = szFileName.find_last_of('.');
38     size_t nLen = szFileName.length();
39     szExtension = szFileName.substr (nLen+1, nLen-nLoc+1);
40 }
41 catch (std::out_of_range e)
42 {
43     std::cout << "string extraction error (OOR): " << e.what()
44             << std::endl;
45 }
46 catch (std::length_error e)
47 {
48     std::cout << "string extraction error (LE): " << e.what()
49             << std::endl;
50 }
51 std::cout << "File extension is " << szExtension << std::endl;
52
53 // Runtime Error 1: Out of range error
54 std::vector<int> V(4);
55 V[0] = 1; V[1] = 2; V[2] = 3; V[3] = 4;
56 size_t nLast = V.size();
57 try
58 {
59     std::cout << "Value at last position is "
60             << V.at(nLast) << std::endl;
61 }
62 catch (std::exception &e)
63 {
64     std::cerr << "Caught " << e.what() << std::endl;
65     std::cerr << "Type " << typeid(e).name() << std::endl;
66 }
67
68 return 0;
69 }
```

Figure 9.5.1 shows the output from running the program.

While the idea of exception handling is good, it is not clear at this stage whether exception handling improves code generation and readability. Exception handling is not implemented uniformly across compilers and operating systems. Hopefully the situation will improve in the near future.

```

Allocated : 1024 DP words
Allocated : 2048 DP words
Allocated : 4096 DP words
Allocated : 8192 DP words
Allocated : 16384 DP words
Allocated : 32768 DP words
Allocated : 65536 DP words
Allocated : 131072 DP words
Allocated : 262144 DP words
Allocated : 524288 DP words
Allocated : 1048576 DP words
Allocated : 2097152 DP words
Allocated : 4194304 DP words
Allocated : 8388608 DP words
Allocated : 16777216 DP words
Allocated : 33554432 DP words
Allocated : 67108864 DP words
Allocated : 134217728 DP words
Cannot allocate : 268435456 DP words
string extraction error (OOR): invalid string position
File extension is ?
Caught invalid vector<T> subscript
Type class std::out_of_range
Press any key to continue . .

```

Fig. 9.5.1 Generated output from MS VS2005 C++ compiler generated program

9.6 Command Line Arguments

So far we have assumed that a typical main program is

```

int main () // or int main (void)
{
.... // one or more return statements returning an integer value
}

```

C/C++ standards specify that console programs containing the `main` program can have arguments passed to it. A typical main program has the following structure.

```

int main (int argc, char *argv[])
{
.... // one or more return statements returning an integer value
}

```

where `argc` is the number of command line arguments and `argv` is a character array containing the command line arguments. Consider the following scenario - you have created a console application `search` and you wish to specify the file from which to search for a specific string. Now assume that you launch the program from command line as follows

```
search address.dat Iowa
```

With this example, argc is 3 – there are 3 command line arguments. The first argument (contained in argv[0]) is `search`, the second argument (contained in argv[1]) is `address.dat` and the last argument is `Iowa` (contained in argv[2]).

Example Program 9.6.1

We will create a sample program called Example9_6_1 and pass it two additional arguments.

main.cpp

```

1 #include <iostream>
2
3 int main (int argc, char* argv[])
4 {
5     std::cout << "Number of command line arguments: "
6         << argc << std::endl;
7
8     std::cout << "\nCommand line arguments are as follows\n";
9     for (int i=0; i < argc; i++)
10    {
11        std::cout << i << ":" << argv[i] << std::endl;
12    }
13    std::cout << std::endl;
14
15    return 0;
16 }
```

The program is simple and self-explanatory. It should be clear that the minimum value of `argc` is 1 so that at least the program name is available in `argv[0]`. One or more blank spaces separate one argument from the next. The sample output is shown in Fig. 9.6.1.

Fig. 9.6.1 Sample output from the program

Summary

The second more advanced look at classes and objects begins to show the strength of OOP and C++. Ideas associated with the `this` pointer, proper use of `const` qualifier, friend classes, use of the reference operator with functions, operator overloading, template classes and especially, the development of two classes to manage vectors and two-dimensional matrices were studied. These classes will prove to be indispensable in handling numerical analysis algorithms.

Programming Style Tip 9.1: Pass objects including arrays by reference

To avoid making a copy of the object being passed, especially if the object is complex and resource hungry, pass objects by reference. For example, The `AddMatrices` function uses pass-by-reference technique (lines 20 and 21). This process is preferable since a local copy is not created if passed by reference. Making a local copy can be time-consuming if the size of the matrix is large. If the matrix should not be modified, use the `const` qualifier.

Programming Style Tip 9.2: Ask yourself if overloaded operators are really necessary?

There is no doubt that the use of overloaded operators in client code makes the code easier to read and hence maintain. However, when resource allocation is an issue, one should be careful in implementing and using overloaded operators. As we saw, sometimes a copy of the object is made during the execution. If this execution is going to consume additional scarce resources, then it may not be a good idea to implement overloaded operators with that class.

Programming Style Tip 9.3: Need for the Big Three

C++ programmers recognize the need for the Big Three – copy constructor, overloaded assignment operator, and destructor, when designing and implementing a class. In other words, if you write your own copy constructor then you should write your own assignment operator and the destructor. The default functionality provided by the C++ compiler (in the absence of your version) may not provide the exact functionality that is necessary for a robust, efficient and correct code.

EXERCISES

Most of the problems below involve the development of one or more classes. In each case develop (a) plan to test the classes(s), and (b) implement the plan in a [main](#) program.

Appetizers

Problem 9.1

Problem 7.4 dealt with the development and implementation of the `CFraction` class. Now enhance the capabilities of the class by overloading the following operators. `F1`, `F2` and `F3` are `CFraction` objects.

```

F1 = F2;           // overloaded operator =
F3 = F1 + F2;    // overloaded operator +
F3 = F1 - F2;    // overloaded operator -
F3 = F1 * F2;    // overloaded operator *
F3 = F1 / F2;    // overloaded operator /
if (F1 == F2)    // overloaded operator ==
if (F1 != F2)    // overloaded operator !=
cout << F3 << "\n"; // overloaded operator <<
cin  >> F3 ;      // overloaded operator >>

```

Problem 9.2

Convert the statistical functions discussed in Example 4.4.1 into member functions of a `CStatistics` class. However, use the `CVector` class to handle vectors instead of C++ arrays. This way you will be able to deal with any number of data values.

Write your main program in such a way that you have a mini-statistical package to support the functionalities discussed in the example. In other words, the program you develop should ask the user for the number of data values, allocate the memory dynamically to store those values, and then call the `CStatistics` member functions to compute all the statistical values and display them on the screen. Set up the program for one time execution only.

Problem 9.3

Change the `CStatistics` class developed in Problem 9.2 to a template class.

Main Course

Problem 9.4

The differential equation for a transverse deflection of a beam of length L subjected to an arbitrary loading, $w(x)$ is given by $\frac{d^4y}{dx^4} = \frac{w(x)}{EI}$. Take $w(x)$ as a uniform load, w .

(a) The solution for a simply-supported beam is given as

$$y(x) = -\frac{wx}{24EI} (x^3 - 2Lx^2 + L^3)$$

(b) The solution for a fixed-fixed beam is given as

$$y(x) = -\frac{wx^2}{24EI} (x^2 - 2Lx + L^2)$$

(c) The solution for a cantilever beam is given as

$$y(x) = -\frac{wx^2}{24EI} (x^2 - 4Lx + 6L^2)$$

Obtain the values of L, E, I, w and the units for force and length from the user. Divide the length of the beam into 20 equally spaced points. Display a table on the screen. The table should have four columns – location and corresponding displacement for the three beam types. Store the table data in a matrix (use the **CMatrix** class).

C++ Concepts

Problem 9.5

Complex numbers are used a variety of engineering and scientific calculations. A complex number z is written as

$$z = x + iy$$

The operations between two complex numbers z_1 and z_2 can be expressed as

$$z_1 + z_2 = (x_1 + x_2) + i(y_1 + y_2)$$

$$z_1 - z_2 = (x_1 - x_2) + i(y_1 - y_2)$$

$$z_1 z_2 = (x_1 x_2 - y_1 y_2) + i(x_1 y_2 + x_2 y_1)$$

$$\frac{z_1}{z_2} = \left(\frac{x_1 x_2 + y_1 y_2}{x_2^2 + y_2^2} \right) + i \left(\frac{x_2 y_1 - x_1 y_2}{x_2^2 + y_2^2} \right)$$

$$|z| = |x + iy| = \sqrt{x^2 + y^2}$$

C++ does not have a class to handle complex numbers³. Implement the `CComplex` class whose class definition is shown below.

```
#ifndef __COMPLEX_H__
#define __COMPLEX_H__

#include <iostream>

class CComplex
{
public:
    CComplex ();
    CComplex (double dR, double dI);
    CComplex (const CComplex&);
    ~CComplex ();
    friend std::ostream &operator<< (std::ostream&, const CComplex&);
    friend double fabs (const CComplex&);

    // accessor functions
    double Real ()const;
    double Imaginary ()const;
    void GetValues (double&, double&) const;
    double Modulus ()const;

    // modifier functions
    void Real (double);
    void Imaginary (double);
    void SetValues (double, double);

    // overloaded operators
    CComplex operator+ (const CComplex& dR) const;
    CComplex& operator+= (const CComplex& dR);
    CComplex operator- (const CComplex& dR) const;
    CComplex& operator-= (const CComplex& dR);
    CComplex operator* (const CComplex& dR) const;
    CComplex& operator*= (const CComplex& dR);
    CComplex operator/ (const CComplex& dR) const;
    CComplex& operator/= (const CComplex& dR);
    CComplex& operator= (const CComplex& dR);
    CComplex& operator= (const double);
    CComplex operator- () const;

private:
    double m_dReal;
    double m_dImaginary;
};

#endif
```

Problem 9.6

³ STL does have a complex class that can be accessed via #include <complex>

Modify the class CGaussQuad to support integration over triangular domains using area coordinates.

Chapter

10

Matrix Algebra

"The person who knows "how" will always have a job. The person who knows "why" will always be his boss." Diane Ravitch

'Real learning comes about when the competitive spirit has ceased.' J. Krishnamurti

In Chapter 6 we saw the solution of an equation in a single unknown. The challenge was to find the roots of a nonlinear equation. More often than not, engineering problems are described by several (hundreds perhaps million) equations. These equations are usually linear in nature or can be approximated as such. In this chapter, we will start by first reviewing the basics of matrix algebra – vector and matrix operations. With that background, we will start to look at different methods to obtain the solution of linear algebraic equations. Finally, we will be in a position to develop an object-oriented matrix toolbox based on the CVector and CMatrix classes developed in Chapter 9. This toolbox will be used in almost all the later chapters dealing with numerical analysis.

Objectives

- To understand matrix algebra.
- To understand the role of matrix algebra in engineering and scientific analysis.
- To understand and implement the steps to solve a system of linear algebraic equations.
- To understand and implement a template matrix toolbox.

10.1 Fundamentals of Matrix Algebra

The solution methodology to be discussed in the chapter is numerical in nature. The initial, intermediate and final steps usually involve matrices. While knowledge of linear algebra is essential in understanding the material in this chapter, we will focus on a narrower topic – matrix algebra.

10.1.1 DEFINITIONS

Matrix: A two-dimensional matrix is a rectangular array of numbers. Each number or element of the matrix is identified by its location – a row number and a column number. Consider the following example.

$$\mathbf{A}_{m \times n} = \begin{bmatrix} A_{11} & A_{12} & \dots & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & \dots & A_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ A_{m1} & A_{m2} & \dots & \dots & A_{mn} \end{bmatrix} \quad (10.1.1-1)$$

A typical element of the matrix \mathbf{A} is designated A_{ij} where i is the row number and j is the column number. We will usually, but not always, denote matrices with an upper case alphabet.

Vector: A vector is a special instance of a matrix. It has either one row or one column. We will usually, but not always, denote a vector with a lower case alphabet.

Row Vector: A vector with one row is called a row vector. Consider the following example.

$$\mathbf{a}_{1 \times n} = \{a_1 \ a_2 \ \dots \ a_n\} \quad (10.1.1-2)$$

Column Vector: A vector with one column is called a column vector. Consider the following example.

$$a_{m \times 1} = \begin{Bmatrix} a_1 \\ a_2 \\ \dots \\ a_m \end{Bmatrix} \quad (10.1.1-3)$$

Null Vector: A null vector is such that all the elements of the vector are zero. For example

$$a_{1 \times n} = \{0 \ 0 \ \dots \ 0\} \quad (10.1.1-4)$$

Square Matrix: A square matrix has the same number of rows and columns. For example,

$$\mathbf{B}_{3 \times 3} = \begin{bmatrix} 12 & -3 & 1 \\ 5 & 8 & 0 \\ -55 & 1 & 22 \end{bmatrix}$$

is a square matrix with integer elements.

Symmetric Matrix: A square matrix such that $A_{ij} = A_{ji}$ for any i, j is a symmetric matrix. For example,

$$\mathbf{B}_{3 \times 3} = \begin{bmatrix} 12 & -3 & 1 \\ -3 & 8 & 0 \\ 1 & 0 & 22 \end{bmatrix}$$

is a symmetric matrix.

Diagonal Matrix: A square matrix such that $A_{ij} = 0$ if $i \neq j$ is a diagonal matrix. For example,

$$\mathbf{B}_{3 \times 3} = \begin{bmatrix} 12 & 0 & 0 \\ 0 & 8 & 0 \\ 0 & 0 & 22 \end{bmatrix}$$

is a diagonal matrix.

Identity Matrix: A diagonal matrix such that $A_{ii} = 1$, $A_{ij} = 0, i \neq j$ is an identity matrix and is denoted $\mathbf{I}_{n \times n}$. For example, the following is an identity (or, unit) matrix of order or size 3.

$$\mathbf{I}_{3 \times 3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Upper Triangular Matrix: A square matrix such that $A_{ij} = 0$ if $i < j$ is an upper triangular matrix. For example,

$$\mathbf{B}_{3 \times 3} = \begin{bmatrix} 12 & -55 & 0 \\ 0 & 8 & 10 \\ 0 & 0 & 22 \end{bmatrix}$$

is an upper triangular matrix.

Lower Triangular Matrix: A square matrix such that $A_{ij} = 0$ if $i > j$ is an lower triangular matrix. For example,

$$\mathbf{B}_{3 \times 3} = \begin{bmatrix} 12 & 0 & 0 \\ -55 & 8 & 0 \\ 0 & 10 & 22 \end{bmatrix}$$

is a lower triangular matrix.

Positive Definite Matrix: A square matrix such that all its eigenvalues are positive. We will look at eigenvalues in Chapter 16.

Orthogonal Matrix: A square matrix such that its transpose is equal to its inverse. In other words, $\mathbf{A}^T \mathbf{A} = \mathbf{A} \mathbf{A}^T = \mathbf{I}$.

Hermitian or Self-Adjoint Matrix: A square matrix such that it is equal to its complex-conjugate of its transpose. In other words, $\mathbf{A} = \mathbf{A}^\dagger$. For a real matrix, Hermitian means the same as symmetric.

10.1.2 OPERATIONS

Addition and Subtraction: Two matrices of the same size can be added or subtracted from one another. For example, if

$$\mathbf{A}_{m \times n} = \mathbf{B}_{m \times n} + \mathbf{C}_{m \times n} \text{ then } A_{ij} = B_{ij} + C_{ij} \quad (10.1.2-1)$$

and, if

$$\mathbf{A}_{m \times n} = \mathbf{B}_{m \times n} - \mathbf{C}_{m \times n} \text{ then } A_{ij} = B_{ij} - C_{ij} \quad (10.1.2-2)$$

Consider the following example. Let

$$\mathbf{B}_{3 \times 3} = \begin{bmatrix} 12 & -3 & 1 \\ -3 & 8 & 0 \\ 1 & 0 & 22 \end{bmatrix} \text{ and } \mathbf{C}_{3 \times 3} = \begin{bmatrix} 0 & 12 & -1 \\ 15 & 8 & 1 \\ 11 & 0 & 7 \end{bmatrix}$$

Then

$$\mathbf{A} = \mathbf{B} + \mathbf{C} = \begin{bmatrix} 12 & 9 & 0 \\ 12 & 16 & 1 \\ 12 & 0 & 29 \end{bmatrix} \text{ and } \mathbf{A} = \mathbf{B} - \mathbf{C} = \begin{bmatrix} 12 & -15 & 2 \\ -18 & 0 & -1 \\ -10 & 0 & 15 \end{bmatrix}$$

Multiplication: Two matrices can be multiplied as follows

$$\mathbf{A}_{m \times n} = \mathbf{B}_{m \times o} \mathbf{C}_{o \times n} \quad (10.1.2-3)$$

provided the number of columns in \mathbf{B} is equal to the number of rows in \mathbf{C} . This condition makes the two matrices conformable. The resulting matrix \mathbf{A} has its number of rows equal to the number of rows in \mathbf{B} and number of columns equal to the number of columns in \mathbf{C} . To generate the elements of the resulting matrix \mathbf{A} we need

$$A_{ij} = \sum_{k=1}^o B_{ik} C_{kj} \quad (10.1.2-4)$$

In other words, the product of the corresponding elements from row i of \mathbf{B} with the elements from column j of \mathbf{C} yields A_{ij} . This operation is similar to computing the dot product.

For example, let

$$\mathbf{B}_{3 \times 3} = \begin{bmatrix} 12 & -3 & 1 \\ -3 & 8 & 0 \\ 1 & 0 & 22 \end{bmatrix} \text{ and } \mathbf{C}_{3 \times 2} = \begin{bmatrix} 0 & 12 \\ 15 & 8 \\ 11 & 0 \end{bmatrix}$$

Then $\mathbf{A}_{3 \times 2} = \mathbf{B}_{3 \times 3} \mathbf{C}_{3 \times 2}$ can be computed by writing the three matrices as follows.

$$\begin{array}{c|c} 0 & 12 \\ \hline 15 & 8 \\ \hline 11 & 0 \end{array} = \begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \\ \hline A_{31} & A_{32} \end{array}$$

$$\begin{bmatrix} 12 & -3 & 1 \\ -3 & 8 & 0 \\ 1 & 0 & 22 \end{bmatrix}$$

where $A_{11} =$ the product of the first row of \mathbf{B} times the first column of \mathbf{C}

$$= (12)(0) + (-3)(15) + (1)(11) = -34$$

A_{12} = the product of the first row of \mathbf{B} times the second column of \mathbf{C}

$$= (12)(12) + (-3)(8) + (1)(0) = 120$$

A_{21} = the product of the second row of \mathbf{B} times the first column of \mathbf{C}

$$= (-3)(0) + (8)(15) + (0)(11) = 120$$

A_{22} = the product of the second row of \mathbf{B} times the second column of \mathbf{C}

$$= (-3)(12) + (8)(8) + (0)(0) = 28$$

A_{31} = the product of the third row of \mathbf{B} times the first column of \mathbf{C}

$$= (1)(0) + (0)(15) + (22)(11) = 242$$

A_{32} = the product of the third row of \mathbf{B} times the second column of \mathbf{C}

$$= (1)(12) + (0)(8) + (22)(0) = 12$$

Transpose: The transpose of matrix $\mathbf{A}_{m \times n}$ is denoted $\mathbf{A}_{n \times m}^T$. The transpose matrix is constructed such that

$$A_{ij}^T = A_{ji} \quad (10.1.2-5)$$

As can be seen from Eqn. (6.1.2-5), the transpose matrix is obtained by interchanging the rows and columns of the original matrix. Let

$$\mathbf{C}_{3 \times 2} = \begin{bmatrix} 0 & 12 \\ 15 & 8 \\ 11 & 0 \end{bmatrix}. \text{ Then } \mathbf{C}_{2 \times 3}^T = \begin{bmatrix} 0 & 15 & 11 \\ 12 & 8 & 0 \end{bmatrix}$$

Determinant: The determinant of a square matrix $\mathbf{A}_{n \times n}$ is denoted $\det(\mathbf{A})$ and is given by

$$\det(\mathbf{A}) = \sum_{j=1}^n a_{ij} A_{ij} \text{ for any } i = 1, 2, \dots, n$$

(10.1.2-6a)

$$\text{or, } \det(\mathbf{A}) = \sum_{i=1}^n a_{ij} A_{ij} \text{ for any } j = 1, 2, \dots, n \quad (10.1.2-6b)$$

where minor M_{ij} is the determinant of the $(n-1) \times (n-1)$ submatrix obtained by deleting the i^{th} row and j^{th} column, and cofactor a_{ij} associated with M_{ij} is defined to be $a_{ij} = (-1)^{i+j} M_{ij}$. While it will not be necessary for us to compute the determinant of a matrix, we still need to understand the concept. Let

$$\mathbf{A}_{2 \times 2} = \begin{bmatrix} 4 & -3 \\ 1 & 6 \end{bmatrix} \text{ and } \mathbf{B}_{2 \times 2} = \begin{bmatrix} 8 & 3 \\ 16 & 6 \end{bmatrix}$$

Then using Eqn. (10.1.2-6a) with $i = 1$,

$$\det(\mathbf{A}) = \sum_{j=1}^n a_{1j} A_{1j} = a_{11} A_{11} + a_{12} A_{12} = 4a_{11} - 3a_{12}$$

$$a_{11} = (-1)^{1+1} \det[6] = (1)(6) = 6 \qquad a_{12} = (-1)^{1+2} \det[1] = (-1)(1) = -1$$

$$\det(\mathbf{A}) = 4a_{11} - 3a_{12} = 4(6) - 3(-1) = 27$$

$$\text{and, } \det(\mathbf{B}) = \sum_{j=1}^n b_{1j} B_{1j} = b_{11} B_{11} + b_{12} B_{12} = 8b_{11} + 3b_{12}$$

$$b_{11} = (-1)^{1+1} \det[6] = (1)(6) = 6 \qquad b_{12} = (-1)^{1+2} \det[16] = (-1)(16) = -16$$

$$\det(\mathbf{B}) = 8b_{11} + 3b_{12} = 8(6) + 3(-16) = 0$$

Since the determinant of \mathbf{B} is zero, \mathbf{B} is known as a *singular* matrix. In the next section we will see a more efficient way to compute the determinant of a matrix.

10.2 Linear Algebraic Equations

Several engineering and scientific problems require the solution of linear algebraic equations of the form

$$\mathbf{A}_{m \times n} \mathbf{x}_{n \times 1} = \mathbf{b}_{m \times 1} \quad (10.2.1)$$

where $\mathbf{A}_{m \times n}$ is the coefficient matrix with constant coefficients, $\mathbf{b}_{m \times 1}$ is the right-hand side (RHS) vector and $\mathbf{x}_{n \times 1}$ is the vector of unknowns. A unique, nontrivial solution, $\mathbf{x}_{n \times 1}$ can be obtained if and only if

- (a) the number of unknowns, n is equal to the number of equations, m such that \mathbf{A} is a square matrix,
- (b) $\mathbf{A}_{n \times n}$ is nonsingular, and
- (c) $\mathbf{b}_{n \times 1}$ is not a null vector (there is at least one non-zero term in \mathbf{b}).

If $m > n$, we have an over determined system. On the other hand, if $m < n$ then it is possible that there are no unique solutions.

Equation solvers can be categorized as being either direct or iterative. Direct solvers are those that solve Eqns. (10.2.1) in a non-iterative fashion. The procedure typically involves one pass through the n equations. On the other hand, iterative solvers transform the problem into an equivalent problem and the solution is improved iteratively.

There are at least four major issues in solving Eqn. (10.2.1).

- (a) How much of storage space will be used especially by the coefficient matrix?
- (b) How can numerically accurate solutions be obtained?
- (c) How much time will be taken to obtain the solution?
- (d) How much of additional effort is needed if a solution is to be generated for a new right-hand side vector?

There are other issues such as parallelizing and vectorizing the solution procedure on specialized hardware and software systems but they are beyond the scope of the discussions here.

10.2.1 STORAGE SCHEME

The coefficient matrix $\mathbf{A}_{n \times n}$ can take on many different forms depending on the application. The matrix can be full meaning that all the elements in the matrix are nonzero. At the other end of the spectrum the matrix can be sparse. For example, solution of some of the popular partial differential equations involves solving a system of equations where the nonzero entries make up a few percent (1-10%) of the entire matrix. The coefficient matrix can have other forms and properties – symmetric, banded, anti-symmetric, skyline, positive definite, indefinite, etc. The implementation efficiency of an algorithm can be tied to its storage scheme.

Full: This is the simplest storage form where the matrix can be stored either rowwise or columnwise. If $\mathbf{A}_{n \times n}$ is stored rowwise then the elements of the matrix are stored as

$$\{A_{11}, A_{12}, \dots, A_{1n}, A_{21}, A_{22}, \dots, A_{2n}, \dots, A_{n1}, A_{n2}, \dots, A_{nn}\}$$

On the other hand if $\mathbf{A}_{n \times n}$ is stored columnwise then the elements of the matrix are stored as

$$\{A_{11}, A_{21}, \dots, A_{n1}, A_{12}, A_{22}, \dots, A_{n2}, \dots, A_{1n}, A_{2n}, \dots, A_{nn}\}$$

In FORTRAN, matrix elements are stored columnwise. In statically allocated C++ matrices, matrix elements are stored rowwise. The `CMatrix` class that we developed in Chapter 9 is designed to store the elements rowwise and with a little tweaking can also be designed to store the elements columnwise. Either way the total storage requirement is n^2 locations.

Banded and Skyline: In the context of finite element analysis, $\mathbf{A}_{n \times n}$ is mostly symmetric and positive definite. Under certain scenarios, \mathbf{A} has a special form that can be exploited from a storage perspective. Consider the symmetric matrix shown in Fig. 10.2.1. The figure shows only the nonzero upper triangular components.

$$\left[\begin{array}{ccccccccc} A_{11} & A_{12} & & A_{15} & A_{16} & & & & \\ & A_{22} & & A_{25} & A_{26} & & & & \\ & & A_{33} & A_{34} & A_{35} & A_{36} & & & \\ & & & A_{44} & A_{45} & A_{46} & & & \\ & & & & A_{55} & A_{56} & A_{57} & A_{58} & A_{59} & A_{5,10} \\ & & & & & A_{66} & A_{67} & A_{68} & A_{69} & A_{6,10} \\ & & & & & & A_{77} & A_{78} & & \\ & & & & & & & A_{88} & & \\ & & & & & & & & A_{99} & A_{9,10} \\ & & & & & & & & & A_{10,10} \end{array} \right]$$

Fig. 10.2.1 Upper triangular portion of the system stiffness matrix

We will draw a special box that encompasses all the nonzero elements in the upper triangular portion of the matrix. As can be seen from Fig. 10.2.2, the nonzero elements are all contained within a band. The width of this band is known as the half-band width (HBW) of the matrix. To find the HBW, we scan each row of the matrix starting with the diagonal element and look for the last nonzero entry in that row. The maximum distance from the last nonzero element to the diagonal element (in a particular row) for all the rows gives the HBW. The formula for HBW of row i is given as

$$(HBW)_i = c - i + 1 \quad (10.2.2a)$$

where c is the column number of the last nonzero element in row i (note, the +1 is used so that the distance includes both the last nonzero element and the diagonal element) and

$$HBW = \max_i (HBW)_i \quad (10.2.2b)$$

For example, in row 1, since the last nonzero element is A_{16} , this distance is $6 - 1 + 1 = 6$. As can be deduced from Fig. 10.2.2, the half-band width (HBW) of the given matrix is 6.

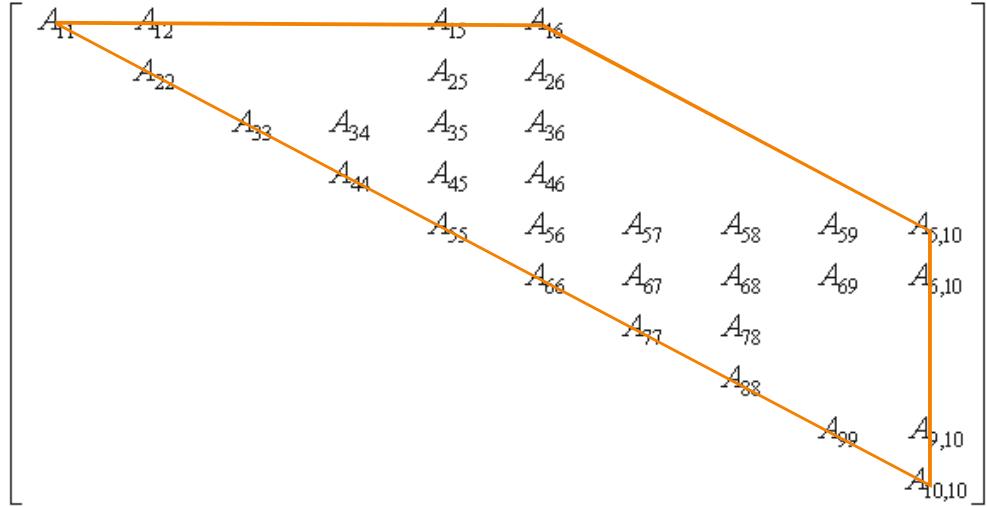


Fig. 10.2.2 Banded profile for the nonzero elements

Instead of storing the entire matrix (full storage), we can store this symmetric, banded matrix as a rectangular matrix as follows.

$$\mathbf{A}_{10 \times 6}^{banded} = \begin{bmatrix} A_{11} & A_{12} & 0 & 0 & A_{15} & A_{16} \\ A_{22} & 0 & 0 & A_{25} & A_{26} & 0 \\ A_{33} & A_{34} & A_{35} & A_{36} & 0 & 0 \\ A_{44} & A_{45} & A_{46} & 0 & 0 & 0 \\ A_{55} & A_{56} & A_{57} & A_{58} & A_{59} & A_{5,10} \\ A_{66} & A_{67} & A_{68} & A_{69} & A_{6,10} & 0 \\ A_{77} & A_{78} & 0 & 0 & 0 & 0 \\ A_{88} & 0 & 0 & 0 & 0 & 0 \\ A_{99} & A_{9,10} & 0 & 0 & 0 & 0 \\ A_{10,10} & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Fig. 10.2.3 Symmetric, banded matrix stored as a rectangular matrix

The relationship (mapping) between the elements in the original \mathbf{A} in Fig. 10.2.1 and the banded form \mathbf{A}^{banded} in Fig. 10.2.3 can be derived simply as follows.

$$A_{i,j} = 0 \text{ if } (j - i + 1) > HBW \quad (10.2.3)$$

$$A_{i,j} = 0 \text{ if } (j < i) \quad (10.2.4)$$

$$\text{else } A_{i,j} \Rightarrow A_{i,j-i+1}^{banded} \quad (10.2.5)$$

As we can see in Fig. 10.2.2, a good number of the elements within the band are zero! We can capitalize even more on this characteristic by storing only the elements within the skyline profile that is shown in Fig. 10.2.4.

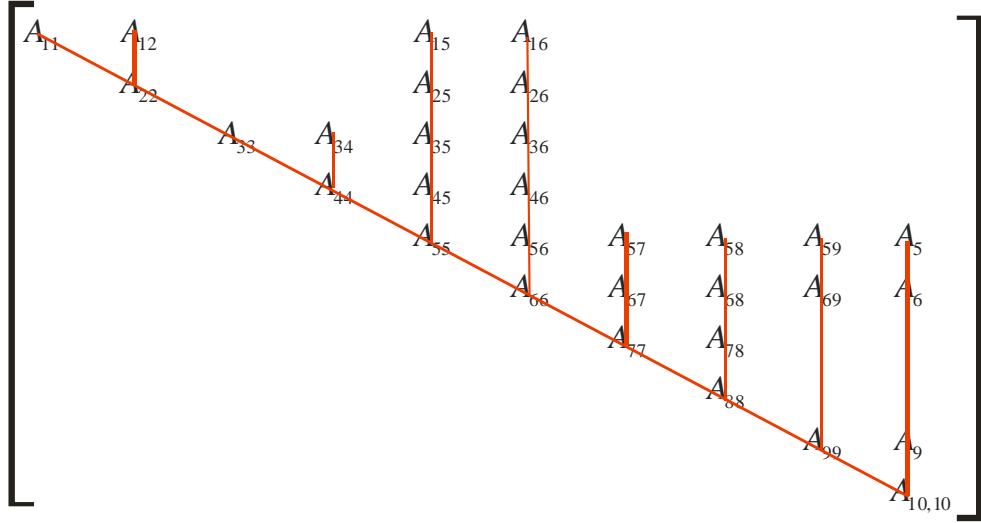


Fig. 10.2.4 The “skyline” profile

The original matrix, in fact, can be stored in a vector as follows.

$$\mathbf{A}_{35 \times 1}^{skyline} = \{A_{11}, A_{22}, A_{12}, A_{33}, A_{44}, A_{34}, \dots, A_{10,10}, A_{9,10}, A_{8,10}, A_{7,10}, A_{6,10}, A_{5,10}\} \quad (10.2.6)$$

Note that each column is stored starting with the diagonal element of that column followed by all the other elements in that column until the last nonzero entry in that column. To facilitate mapping the original elements of the matrix, an additional indexing vector, \mathbf{D}_{n+1}^{loc} is created that has $(n + 1)$ elements. These elements store the location of the diagonal element (of each column) with the last element storing the (last element+1) in the stiffness matrix. In other words, the last element contains one more than the total number of entries in the skyline profile. Going back to the current example, we have the following \mathbf{D}_{n+1}^{loc} vector.

$$\mathbf{D}_{11}^{loc} = \{1, 2, 4, 5, 7, 12, 18, 21, 25, 30, 36\} \quad (10.2.7)$$

The relationship (mapping) between the elements in the original \mathbf{A} in Fig. 10.2.2 and the skyline form $\mathbf{A}^{skyline}$ in Fig. 10.2.4 can be derived as follows.

$$A_{i,j} = 0 \text{ if } (j-i) > (D_{j+1}^{loc} - D_j^{loc}) \quad (10.2.8)$$

$$A_{i,j} = 0 \text{ if } (j < i) \quad (10.2.9)$$

$$A_{i,j} \Rightarrow l = D_j^{loc} + j - i \Rightarrow A_l^{skyline} \quad (10.2.10)$$

For example, to locate A_{68} we note that (a) $i = 6, j = 8$, (b) $D_8^{loc} = 21$, (c) $l = 21 + 8 - 6 = 23$. Hence, $A_{68} \Rightarrow A_{23}^{skyline}$.

Finally, let us compare the storage requirements of the three schemes – full, banded, and skyline, assuming that the stiffness matrix is stored in the double precision format, and that two integer words make up a single double precision word ($q = hbw, m = \mathbf{D}_{n+1}^{loc} - 1$).

Storage Scheme	What is to be stored?	Equivalent integer words
Full	$\mathbf{A}_{n \times n}$	$2n^2$
Banded	$\mathbf{A}_{n \times hbw}^{banded}$	$2nq$
Skyline	$\mathbf{A}_m^{skyline}, \mathbf{D}_{n+1}^{loc}$	$2m + n + 1$

Using our example, we have the three values in the last column as 200, 120 and 81 integer words – significant savings with increasing sophistication of the storage scheme.

Sparse: It is not evident with our simple example that the matrix is sparse. The percent sparsity of the matrix is $\frac{35}{100} \times 100 = 35\%$. For some engineering problems, as the size of the problem increases, the sparsity of the matrix increases (or the number of nonzero terms decreases). In one of the sparse storage schemes, three vectors are used to track the locations of the nonzero entries. The matrix is stored in a vector rowwise with only the non-zero entries being stored¹ in $\mathbf{A}_{m \times 1}^{sparse}$. Using our previous example, we have the following.

$$\mathbf{A}_{31 \times 1}^{sparse} = \{A_{11}, A_{12}, A_{15}, A_{16}, A_{22}, A_{25}, A_{26}, \dots, A_{88}, A_{99}, A_{9,10}, A_{10,10}\} \quad (10.2.11)$$

To access the entries in the matrix, two additional (indexing) vectors are needed. The first, $\mathbf{C}_{m \times 1}$ is used to store the column numbers of the nonzero entries. Again, we have with our example

¹ It should be noted that zero entries within the skyline profile may become nonzero during the solution phase.

$$\mathbf{C}_{3 \times 1} = \{1, 2, 5, 6, 2, 5, 6, 3, 4, 5, 6, \dots, 8, 9, 10, 10\} \quad (10.2.12)$$

The second, $\mathbf{R}_{(n+1) \times 1}$, stores the starting location of each row. Again, we have with our example

$$\mathbf{R}_{1 \times 1} = \{1, 5, 8, 12, 15, 21, 26, 28, 29, 31, 32\} \quad (10.2.13)$$

Discussion of sparse equation solvers is outside the scope of this book.

Offline: There are special situations when the solution procedure operates on matrices that are written on and retrieved from computer hard disk. This is typically done when the size of the stiffness matrix and other associated vectors/matrices in any storage format is several times the size of the computer's random access memory (RAM). Discussion of offline equation solvers is outside the scope of this book.

10.2.1 DIRECT SOLVERS

In this section we will see a number of solution techniques starting with the Gaussian Elimination technique.

Gaussian Elimination

There are three operations that can be used on a system of equations to obtain another equivalent system.

- (1) We can interchange the order of two equations.
- (2) Both sides of an equation may be multiplied by a nonzero constant.
- (3) A multiple of one equation can be added to another equation.

Consider the following set of equations.

$$\begin{bmatrix} 8 & -1 \\ 4 & 7 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 6 \\ 18 \end{Bmatrix} \Rightarrow \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 2 \end{Bmatrix}$$

(1) Interchanging the two equations, we have the following set of equations.

$$\begin{bmatrix} 4 & 7 \\ 8 & -1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 18 \\ 6 \end{Bmatrix} \Rightarrow \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 2 \end{Bmatrix}$$

Note that row interchanges do not require changing the order of the unknowns.

(2) We could multiply the second equation by a constant (say 1.5) to obtain the following set of equations.

$$\begin{bmatrix} 8 & -1 \\ 6 & 10.5 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 6 \\ 27 \end{Bmatrix} \Rightarrow \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 2 \end{Bmatrix}$$

(3) If we multiply the first equation by $\left(-\frac{4}{8}\right) = -0.5$ and add to the second equation, we obtain the following set of equations.

$$\begin{bmatrix} 8 & -1 \\ 0 & 7.5 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 6 \\ 15 \end{Bmatrix} \Rightarrow \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 2 \end{Bmatrix}$$

The central idea in the Gaussian Elimination technique is based on generating an equivalent set of equations using row operations (3) discussed earlier.

There are two phases to the solution – forward elimination and backward substitution. In the forward elimination phase, the basic idea is to take the set of equations from the original form

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{1i} & A_{1n} \\ A_{21} & A_{22} & A_{23} & A_{2i} & A_{2n} \\ A_{31} & A_{32} & A_{33} & A_{3i} & A_{3n} \\ A_{i1} & A_{i2} & A_{i3} & A_{ii} & A_{in} \\ A_{n1} & A_{n2} & A_{n3} & A_{ni} & A_{nn} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_i \\ x_n \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ b_3 \\ b_i \\ b_n \end{Bmatrix} \quad (10.2.14)$$

to an upper triangular matrix of the form

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{1i} & A_{1n} \\ 0 & A_{22}^{(1)} & A_{23}^{(1)} & A_{2i}^{(1)} & A_{2n}^{(1)} \\ 0 & 0 & A_{33}^{(2)} & A_{3i}^{(2)} & A_{3n}^{(2)} \\ 0 & 0 & 0 & A_{ii}^{(i-1)} & A_{in}^{(i-1)} \\ 0 & 0 & 0 & 0 & A_{nn}^{(n-1)} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_i \\ x_n \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2^{(1)} \\ b_3^{(2)} \\ b_i^{(i-1)} \\ b_n^{(n)} \end{Bmatrix} \quad (10.2.15)$$

With each step through the equations, one unknown is eliminated until only x_n remains as the unknown in the equation. In step k ($k = 1, 2, \dots, n-1$)

$$A_{ij}^{(k)} = A_{ij}^{(k-1)} - \frac{A_{ik}^{(k-1)}}{A_{kk}^{(k-1)}} A_{kj}^{(k-1)} \quad i, j = k+1, \dots, n \quad (10.2.16)$$

$$b_i^{(k)} = b_i^{(k-1)} - \frac{A_{ik}^{(k-1)}}{A_{kk}^{(k-1)}} b_k^{(k-1)} \quad i = k+1, \dots, n \quad (10.2.17)$$

If $|A_{kk}^{(k-1)}| \leq \varepsilon$, where ε is a small positive constant, then the system of equations is linearly dependent.

In the backward substitution phase (dropping the superscript), we first compute the value of the last unknown (x_n) followed by the other unknowns as shown below.

$$x_n = \frac{b_n}{A_{nn}} \quad (10.2.18)$$

$$\text{and } x_i = \frac{b_i - \sum_{j=i+1}^n A_{ij}x_j}{A_{ii}} \quad i = n-1, n-2, \dots, 1 \quad (10.2.19)$$

Algorithm

Step 1: Forward Elimination. Loop through rows, $k = 1, \dots, n-1$.

Step 2: Check if $|A_{kk}| < \varepsilon$. If yes, stop. The equations are linearly dependent (or \mathbf{A} is singular).

Step 3: Loop through columns, $i = k+1, \dots, n$.

Step 4: Compute constant, $c = \frac{A_{ik}}{A_{kk}}$.

Step 5: Loop through $j = k+1, \dots, n$.

Step 6: Set $A_{ij} = A_{ij} - cA_{kj}$.

Step 7: End loop j .

Step 8: Set $b_i = b_i - cb_k$.

Step 9: End loop i .

Step 10: End loop k .

Step 11: Backward substitution. Set $x_n = b_n / A_{nn}$.

Step 12: Loop through all rows, $i = n-1, \dots, 1$.

Step 13: Compute $sum = \sum_{j=i+1}^n A_{ij}x_j$.

Step 14: Compute $x_i = \frac{b_i - sum}{A_{ii}}$.

Step 15: End loop i .

Example 10.2.1

Solve the following set of equations using Gaussian Elimination method.

$$\begin{bmatrix} 10 & -5 & 2 \\ 3 & 20 & 5 \\ -2 & 7 & 15 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 6 \\ 58 \\ 57 \end{Bmatrix}$$

Solution

Forward Substitution

Noting that $n = 3$, the successive snapshots as we go through the algorithm are as follows.

$$k=1 \Rightarrow \begin{bmatrix} 10 & -5 & 2 \\ & 21.5 & 4.4 \\ & 6 & 15.4 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 6 \\ 56.2 \\ 58.2 \end{Bmatrix}$$

$$k=2 \Rightarrow \begin{bmatrix} 10 & -5 & 2 \\ & 21.5 & 4.4 \\ & & 14.172 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 6 \\ 56.2 \\ 42.5163 \end{Bmatrix}$$

Backward Substitution

$$x_3 = \frac{42.5163}{14.172} = 3$$

$$i=2 \Rightarrow x_2 = \frac{56.2 - 4.4(3)}{21.5} = 2$$

$$i=1 \Rightarrow x_1 = \frac{6 - (-4)}{10} = 1$$

Hence the solution is $\mathbf{x}_{3 \times 1} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$.

Pivoting and Scaling: Numerical problems of the form of truncation and round-off errors can be problematic if the coefficient matrix \mathbf{A} is not well-conditioned. A small change in \mathbf{b} resulting in a large change in \mathbf{x} is symptomatic of this ill-conditioning. The culprit in the standard implementation of the Gaussian Elimination technique is A_{kk} that is used in Step 4. With rounding errors, this number could turn out to be very small and using this value could result in gross errors. Pivoting can be used to improve the computations and reduce the effects of the numerical errors.

Partial pivoting: For $1 \leq k \leq n - 1$, at the k^{th} stage, let

$$c_k = \max_{k \leq i \leq n} |A_{ik}^{(k)}| \quad (10.2.20a)$$

Let i be the row index such that $i \geq k$ for which we obtain c_k . If $i > k$, we switch rows k and i in both \mathbf{A} and \mathbf{b} . By using the largest remaining element in \mathbf{A} , we prevent the creation of elements in $\mathbf{A}^{(k)}$ of greatly varying size that leads to numerical errors.

Total pivoting: For $1 \leq k \leq n - 1$, at the k^{th} stage, let

$$c_k = \max_{k \leq i, j \leq n} |A_{ij}^{(k)}| \quad (10.2.20b)$$

Let i, j be the row and column indices such that $i, j \geq k$ for which we obtain c_k . If $i > k$, we switch rows k and i in both \mathbf{A} and \mathbf{b} . If $j > k$, we switch columns k and j in \mathbf{A} and the order of the unknowns in \mathbf{x} . At the end of the solution, we need to switch the order of the unknowns back to the original form.

Numerical studies have shown that total pivoting prevents the catastrophic accumulation of roundoff errors. However, total pivoting is an expensive process. Partial pivoting provides adequate relief for most practical problems.

Scaling: Round-off errors are likely to dominate if the elements of the coefficient matrix \mathbf{A} vary greatly in value. One solution to this problem is to scale \mathbf{A} so that this variation in value is less, and this can be achieved by multiplying the rows and columns by an appropriate constant. In practice, it is necessary to scale the rows so that they are approximately equal in magnitude. Similarly, \mathbf{x} should be scaled so that all the unknowns are approximately equal.

Let \mathbf{S}_1 and \mathbf{S}_2 be diagonal scaling matrices so that

$$\hat{\mathbf{A}} = \mathbf{S}_1 \mathbf{A} \mathbf{S}_2 \quad (10.2.21a)$$

Hence the solution to the original equations can be obtained by

$$\widehat{\mathbf{A}}\mathbf{y} = \mathbf{S}_1 \mathbf{b} \quad \text{and} \quad \mathbf{x} = \mathbf{S}_2 \mathbf{y} \quad (10.2.21b)$$

To select the appropriate scaling constant, it is desirable that

$$\max_{1 \leq j \leq n} |\widehat{A}_{ij}| \approx 1 \quad i = 1, 2, \dots, n \quad (10.2.21c)$$

The selection of the pivot elements for step k is

$$c_k = \max_{k \leq i \leq n} \left| \frac{A_{ik}^{(k)}}{s_i} \right| \quad s_i = \max_{1 \leq j \leq n} |A_{ij}| \quad i = 1, 2, \dots, n \quad (10.2.21d)$$

which is a modification of the condition used in partial pivoting (Eqn. (10.2.20b)). The modified Gaussian Elimination algorithm is presented below. There is an additional vector piv that is needed to store the row interchange information. In other words, if $piv(k) = k$, then no row interchange took place. Otherwise $piv(k) = i$ indicates that rows i and k were interchanged at step k .

Algorithm

Step 1: Forward Elimination. Compute $s_i = \max_{1 \leq j \leq n} |A_{ij}| \quad i = 1, 2, \dots, n$.

Step 2: Loop through rows, $k = 1, \dots, n - 1$.

Step 3: Compute constant, $c_k = \max_{k \leq i \leq n} \left| \frac{A_{ik}}{s_i} \right|$.

Step 4: Let i_0 be the smallest index $i \geq k$ for which the maximum in Step 3 is attained. Store $piv(k) = i_0$.

Step 5: Check if $|c_k| < \varepsilon$. If yes, stop. The equations are linearly dependent (or \mathbf{A} is singular).

Step 6: If $i_0 \neq k$, interchange A_{kj} and $A_{i_0 j}$, $j = k, \dots, n$.

Step 7: Loop through columns, $i = k + 1, \dots, n$.

Step 8: Set $A_{ik} = s = A_{ik} / A_{kk}$.

Step 9: Set $A_{ij} = A_{ij} - sA_{kj}$, $j = k + 1, \dots, n$.

Step 10: End loop i .

Step 10: End loop k .

Step 11: Backward substitution. Loop through $k = 1, 2, \dots, n - 1$.

Step 11: If $j = piv(k) \neq k$, then interchange b_j and b_k .

Step 12: Set $b_i = b_i - A_{ik}b_k \quad i = k + 1, \dots, n$

Step 13: End loop k .

Step 14: Set $b_n = b_n / A_{nn}$.

Step 15: Loop through all rows, $i = n - 1, \dots, 1$.

Step 16: Compute $sum = \sum_{j=i+1}^n A_{ij}x_j$.

Step 17: Compute $x_i = \frac{b_i - sum}{A_{ii}}$.

Step 18: End loop i .

Error Analysis

It is practical and necessary at the end of the solution process to examine the quality of the solution. If \mathbf{x} is indeed the solution, then the residual vector $\mathbf{r} = \mathbf{Ax} - \mathbf{b}$ should be zero. Numerically, the residual vector is not zero and one must ascertain the magnitude of the residual terms. One can define the absolute error and relative errors

$$\varepsilon_{abs} = \|\mathbf{r}\| \quad (10.2.22)$$

$$\varepsilon_{rel} = \frac{\|\mathbf{r}\|}{\|\mathbf{b}\|} \quad (10.2.23)$$

Provided both these measures are small, the solution \mathbf{x} is acceptable.

Example 10.2.2

Solve the following set of equations using Gaussian Elimination method with and without partial pivoting. Compute the residual vector in each case.

$$\begin{bmatrix} 0.7 & 0.8 & 0.9 \\ 1.0000001 & 1.0 & 1.0 \\ 1.3 & 1.2 & 1.1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 0.7 \\ 0.8 \\ 1.0 \end{Bmatrix}$$

Solution

Numerically the coefficient matrix is nearly singular.

The solution and residual vector **using pivoting** is as follows.

$$\mathbf{x} = \begin{Bmatrix} -499999.9994 \\ 1000000.649 \\ -499999.7994 \end{Bmatrix} \quad \mathbf{r} = \begin{Bmatrix} -1.16415(10^{-10}) \\ -5.82076(10^{-11}) \\ 0 \end{Bmatrix}$$

The solution and residual vector **without pivoting** is as follows.

$$\mathbf{x} = \begin{Bmatrix} 0.7 \\ -0.2000001 \\ 0.09999988 \end{Bmatrix} \quad \mathbf{r} = \begin{Bmatrix} -0.280000188 \\ -0.20000015 \\ -0.220000252 \end{Bmatrix}$$

LU Factorization

The matrix \mathbf{A} in $\mathbf{Ax} = \mathbf{b}$ can be factored² as $\mathbf{A} = \mathbf{LU}$ where \mathbf{L} is a lower triangular matrix with nonzero values on the diagonal and below, and \mathbf{U} is an upper triangular matrix with nonzero values on the diagonal and above. For example, we could describe the situation symbolically as follows.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} = \begin{bmatrix} L_{11} & & & \\ L_{21} & L_{22} & & \\ L_{31} & L_{32} & L_{33} & \\ L_{41} & L_{42} & L_{43} & L_{44} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ & U_{22} & U_{23} & U_{24} \\ & & U_{33} & U_{34} \\ & & & U_{44} \end{bmatrix} \quad (10.2.24)$$

The attractive aspect of this procedure is that we can solve the original equations (10.2.1) as follows.

$$\mathbf{A}_{n \times n} \mathbf{x}_{n \times 1} = \mathbf{b}_{n \times 1} \quad (10.2.25a)$$

$$\text{or} \quad \mathbf{L}_{n \times n} \mathbf{U}_{n \times n} \mathbf{x}_{n \times 1} = \mathbf{b}_{n \times 1} \quad (10.2.25b)$$

$$\text{Let} \quad \mathbf{L}_{n \times n} \mathbf{y}_{n \times 1} = \mathbf{b}_{n \times 1} \quad (10.2.25c)$$

and solve for \mathbf{y} . Then solve

² We do not present a formal proof here.

$$\mathbf{U}_{n \times n} \mathbf{x}_{n \times 1} = \mathbf{y}_{n \times 1} \quad (10.2.25d)$$

for \mathbf{x} . The implication is that once \mathbf{L} and \mathbf{U} have been obtained, a new RHS vector requires just forward and backward substitutions in Eqns. (10.2.21c-d). In other words, the forward substitution involves computing

$$y_1 = \frac{b_1}{L_{11}} \quad (10.2.26a)$$

$$y_i = \frac{b_i - \sum_{j=1}^{i-1} L_{ij} y_j}{L_{ii}} \quad i = 2, 3, \dots, n \quad (10.2.26b)$$

Similarly, the backward substitution involves computing

$$x_n = \frac{y_n}{U_{nn}} \quad (10.2.27a)$$

$$x_i = \frac{y_i - \sum_{j=i+1}^n U_{ij} x_j}{U_{ii}} \quad i = n-1, n-2, \dots, 1 \quad (10.2.27b)$$

The major question that remains is how do we obtain the \mathbf{L} and \mathbf{U} matrices? If we multiply the \mathbf{L} and \mathbf{U} matrices, we obtain for a typical term

$$A_{ij} = L_{i1}U_{1j} + L_{i2}U_{2j} + \dots \quad (10.2.28)$$

However, we should note that all elements of \mathbf{L} and \mathbf{U} matrices do not exist. Hence,

$$i < j : \quad A_{ij} = L_{i1}U_{1j} + L_{i2}U_{2j} + \dots + L_{in}U_{nj} \quad (\text{upper triangular elements}) \quad (10.2.29a)$$

$$i = j : \quad A_{ij} = L_{i1}U_{1j} + L_{i2}U_{2j} + \dots + L_{ii}U_{jj} \quad (\text{diagonal elements}) \quad (10.2.29b)$$

$$i > j : \quad A_{ij} = L_{i1}U_{1j} + L_{i2}U_{2j} + \dots + L_{jn}U_{nj} \quad (\text{lower triangular elements}) \quad (10.2.29c)$$

These three sets of equations indicate that we have n^2 equations for $(n^2 + n)$ unknowns – the elements in \mathbf{L} and \mathbf{U} matrices. Crout's Algorithm simply is to set the diagonal entries in \mathbf{L} as unity, i.e. $L_{ii} = 1$. Now we can solve n^2 equations in n^2 unknowns. An examination of the three equations will show that we can start with column 1 and compute U_{11} . Once we know U_{11} , we can compute all the elements in the first column of \mathbf{L} , i.e. $L_{i1}, i = 2, \dots, n$. Next we compute the elements in the second

column of \mathbf{U} , i.e. U_{12}, U_{22} . After that we can compute the elements in the second column of \mathbf{L} , i.e. $L_{i2}, i = 3, \dots, n$. We repeat this process until the elements in all the columns of \mathbf{L} and \mathbf{U} have been computed.

Algorithm

Phase 1: Factorization

Step 1: Loop through all diagonal entries in \mathbf{L} and set them to unity, $L_{ii} = 1, i = 1, 2, \dots, n$.

Step 2: Loop through $j = 1, 2, 3, \dots, n$.

Step 3: Loop through $i = 1, 2, \dots, j$. Set $U_{ij} = A_{ij} - \sum_{k=1}^{i-1} L_{ik} U_{kj}$.

Step 4: Check if $|U_{jj}| < \varepsilon$. If yes, stop. The equations are linearly dependent (or \mathbf{A} is singular).

Step 5: Loop through $i = j+1, j+2, \dots, n$. Set $L_{ij} = \frac{A_{ij} - \sum_{k=1}^{j-1} L_{ik} U_{kj}}{U_{jj}}$.

Step 6: End loop through j .

Phase 2: Forward and backward substitution

Step 1: Solve Eqn. (10.2.25c): Solve for y_1 (Eqn. 10.2.26a). Loop through $i = 2, 3, \dots, n$ and solve for y_i using Eqn. (10.2.26b).

Step 2: Solve Eqn. (10.2.25d): Solve for x_n (Eqn. 10.2.27a). Loop through $i = n-1, n-2, \dots, 1$ and solve for x_i using Eqn. (10.2.27b).

The attractive aspect of this algorithm is that we really do not need (additional) storage space for the \mathbf{L} and \mathbf{U} matrices. The original \mathbf{A} matrix can be overwritten with these matrices such that

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \Rightarrow \begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ L_{21} & U_{22} & U_{23} & U_{24} \\ L_{31} & L_{32} & U_{33} & U_{34} \\ L_{41} & L_{42} & L_{43} & U_{44} \end{bmatrix}$$

Furthermore, we do not need space for \mathbf{y} . We first implement Step 1 in Phase 2 using \mathbf{x} to store the \mathbf{y} values. Then Step 2 can be computed with every successive y_i value replaced with the x_i value starting $y_n \rightarrow x_n$.

The LU Factorization provides a convenient and efficient way to compute the determinant of a matrix. Since $\mathbf{A} = \mathbf{L}\mathbf{U}$, we have $\det(\mathbf{A}) = \det(\mathbf{L})\det(\mathbf{U})$. Since $\det(\mathbf{L}) = 1$, the determinant of the original matrix is simply $\det(\mathbf{A}) = \det(\mathbf{U}) = U_{11}U_{22} \cdots U_{nn}$.

Example 10.2.3

Solve the following set of equations using LU Factorization and compute the determinant of the coefficient matrix.

$$\begin{bmatrix} 10 & -5 & 2 \\ 3 & 20 & 5 \\ -2 & 7 & 15 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 6 \\ 58 \\ 57 \end{Bmatrix}$$

Solution

Factorization ($n = 3$)

$$j = 1, i = 1 \Rightarrow U_{11} = A_{11} = 10$$

$$j = 1, i = 2 \Rightarrow L_{21} = \frac{A_{21}}{U_{11}} = \frac{3}{10} = 0.3$$

$$j = 1, i = 3 \Rightarrow L_{31} = \frac{A_{31}}{U_{11}} = \frac{-2}{10} = -0.2$$

$$j = 2, i = 1 \Rightarrow U_{12} = A_{12} = -5$$

$$j = 2, i = 2 \Rightarrow U_{22} = A_{22} - L_{21}U_{12} = 20 - (0.3)(-5) = 21.5$$

$$j = 2, i = 3 \Rightarrow L_{32} = \frac{A_{32} - L_{31}U_{12}}{U_{22}} = \frac{7 - (-0.2)(-5)}{21.5} = 0.27907$$

$$j = 3, i = 1 \Rightarrow U_{13} = A_{13} = 2$$

$$j = 3, i = 2 \Rightarrow U_{23} = A_{23} - L_{21}U_{13} = 5 - (0.3)(2) = 4.4$$

$$j = 3, i = 3 \Rightarrow U_{33} = A_{33} - L_{31}U_{13} - L_{32}U_{23} = 15 - (-0.2)(2) - (0.27907)(4.4) = 14.1721$$

Hence the new coefficient matrix replaced by the elements of **L** and **U** matrices is as follows.

$$\mathbf{A} = \begin{bmatrix} 10 & -5 & 2 \\ 0.3 & 21.5 & 4.4 \\ -0.2 & 0.27907 & 14.1721 \end{bmatrix}$$

Forward Substitution

$$i = 1 \Rightarrow y_1 = \frac{b_1}{L_{11}} = \frac{6}{1} = 6$$

$$i = 2 \Rightarrow y_2 = \frac{b_2 - L_{21}y_1}{L_{22}} = \frac{58 - (0.3)(6)}{1} = 56.2$$

$$i = 3 \Rightarrow y_3 = \frac{b_3 - L_{31}y_1 - L_{32}y_2}{L_{33}} = \frac{57 - (-0.2)(6) - (0.27907)(56.2)}{1} = 42.5163$$

Backward Substitution

$$i = 3 \Rightarrow x_3 = \frac{y_3}{U_{33}} = \frac{42.5163}{14.1721} = 3$$

$$i = 2 \Rightarrow x_2 = \frac{y_2 - U_{23}x_3}{U_{22}} = \frac{56.2 - (4.4)(3)}{21.5} = 2$$

$$i = 1 \Rightarrow x_1 = \frac{y_1 - U_{12}x_2 - U_{13}x_3}{U_{11}} = \frac{6 - (-5)(2) - (2)(3)}{10} = 1$$

*Determinant of **A***

$$\det(A) = U_{11}U_{22}U_{33} = (10)(21.5)(14.1721) = 3047$$

Cholesky Decomposition or **LDL^T** Factorization (or, Decomposition)

When **A** is symmetric and non-singular, the matrix can be factored as **A** = **LDL^T** where **L** is a lower triangular matrix with 1's on the diagonals and **D** is a diagonal matrix with positive entries. The **LDL^T** decomposition is a variation of Cholesky Decomposition³, and provides a very effective solution to the system equilibrium equations.

³ Strictly speaking, Cholesky Decomposition **A** = **U^TU** can only be used with a symmetric positive definite **A**.

The solution proceeds as follows. We first factor $\mathbf{A} = \mathbf{LDL}^T$ by finding the \mathbf{L} and \mathbf{D} matrices.

$$\begin{bmatrix} K_{11} & K_{12} & \dots & K_{1n} \\ K_{12} & K_{22} & \dots & K_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ K_{1n} & K_{2n} & \dots & K_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ L_{21} & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ L_{n1} & L_{n2} & \dots & 1 \end{bmatrix} \begin{bmatrix} D_1 & 0 & \dots & 0 \\ 0 & D_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & D_n \end{bmatrix} \begin{bmatrix} 1 & L_{21} & \dots & L_{n1} \\ 0 & 1 & \dots & L_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

$$= \left[\begin{array}{c|c|c|c|c} D_1 & D_1 L_{21} & D_1 L_{31} & \dots & D_1 L_{n1} \\ \hline & D_1 L_{21}^2 + D_2 & D_1 L_{21} L_{31} + D_2 L_{32} & \dots & D_1 L_{21} L_{n1} + D_2 L_{n2} \\ \hline & & D_1 L_{31}^2 + D_2 L_{32}^2 + D_3 & \dots & D_1 L_{31} L_{n1} + D_2 L_{32} L_{n2} + D_3 L_{n3} \\ \hline & & & \ddots & \\ \hline sym & & & & D_1 L_{n1}^2 + D_2 L_{n2}^2 + \dots + D_n \end{array} \right] \quad (10.2.30)$$

By comparing the RHS of Eqn. (10.2.30) with the LHS we have the mechanism to compute the \mathbf{L} and \mathbf{D} matrices given the \mathbf{A} matrix.

To obtain the solution once the decomposition or factorization is completed, we have the following steps.

$$\mathbf{LDL}^T \mathbf{x} = \mathbf{b} \quad (10.2.31)$$

$$\text{Let } \mathbf{Ly} = \mathbf{b} \quad (10.2.32)$$

$$\text{Then } \mathbf{DL}^T \mathbf{x} = \mathbf{y} \quad (10.2.33)$$

Note that \mathbf{DL}^T is of the upper triangular form.

$$\mathbf{DL}^T = \begin{bmatrix} D_1 & D_1 L_{12} & \dots & D_1 L_{1n} \\ & D_2 & \dots & D_2 L_{2n} \\ & & \ddots & \vdots \\ \mathbf{0} & & & D_n \end{bmatrix} \quad (10.2.34)$$

We can solve Eqns. (10.2.32) for \mathbf{y} through the forward substitution procedure that we have seen before. Once \mathbf{y} has been computed, we can solve Eqns. (10.2.33) through the backward substitution process.

Algorithm

Factorization Phase

Step 1: Loop through rows, $i = 1, \dots, n$.

Step 2: Set $D_i = A_{ii} - \sum_{j=1}^{i-1} L_{ij}^2 D_j$. If $D_i < \varepsilon$, stop. The matrix is not positive definite.

Step 3: For $j = i+1, \dots, n$, set $L_{ji} = \frac{A_{ji} - \sum_{k=1}^{i-1} L_{jk} D_k L_{ik}}{D_i}$.

Step 4: End loop i .

Forward and Backward Substitutions

Step 5: Forward Substitution. Set $y_1 = b_1$.

Step 6: For $i = 2, \dots, n$, set $y_i = b_i - \sum_{j=1}^{i-1} L_{ij} y_j$. This ends the Forward Substitution phase.

Step 7: Backward Substitution. Set $x_n = \frac{y_n}{D_n}$.

Step 8: For $i = n-1, \dots, 1$, set $x_i = \frac{y_i}{D_i} - \sum_{j=i+1}^n L_{ji} x_j$. This ends the Backward Substitution phase.

A careful examination of the steps will show that no extra storage is required. The storage locations in **A** can be used to store both **D** and **L**. Similarly, the storage locations in **x** can be used to store the elements of **y**.

Similar to LU Decomposition, Cholesky Decomposition requires no special effort to solve additional RHS vectors since the factorization and the forward/backward substitutions steps are separate. Once the factorization step is completed (**once**), the forward/backward substitutions steps can be repeated as many times as required.

Example 10.2.4

Solve the following set of equations using Cholesky Decomposition method.

$$\begin{bmatrix} 3.5120 & 0.7679 & 0 & 0 & 0 \\ 0.7679 & 3.1520 & 0 & -2 & 0 \\ 0 & 0 & 3.5120 & -0.7679 & 0.7679 \\ 0 & -2 & -0.7679 & 3.1520 & -1.1520 \\ 0 & 0 & 0.7679 & -1.1520 & 3.1520 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ -0.04 \\ 0 \end{Bmatrix}$$

Solution

Factorization (n = 5)

$$i=1 \Rightarrow D_1 = A_{11} = 3.5120.$$

$$j=2 \Rightarrow L_{21} = \frac{A_{21}}{D_1} = \frac{0.7679}{3.5120} = 0.21865. \text{ Also, } L_{31} = L_{41} = L_{51} = 0.$$

$$i=2 \Rightarrow D_2 = A_{22} - L_{21}^2 D_1 = 3.1520 - (0.21865)^2 (3.5120) = 2.9841.$$

$$j=3 \Rightarrow L_{32} = \frac{A_{32} - L_{31} D_1 L_{21}}{D_2} = 0.$$

$$j=4 \Rightarrow L_{42} = \frac{A_{42} - L_{41} D_1 L_{21}}{D_2} = \frac{-2 - 0}{2.9841} = -0.670219. \text{ Also, } L_{52} = 0.$$

$$i=3 \Rightarrow D_3 = A_{33} - L_{31}^2 D_1 - L_{32}^2 D_2 = 3.5120 - 0 - 0 = 3.5120.$$

$$j=4 \Rightarrow L_{43} = \frac{A_{43} - L_{41} D_1 L_{31} - L_{42} D_2 L_{32}}{D_3} = \frac{-0.7679 - 0 - 0}{3.5120} = -0.21865.$$

$$j=5 \Rightarrow L_{53} = \frac{A_{53} - L_{51} D_1 L_{31} - L_{52} D_2 L_{32}}{D_3} = \frac{0.7679 - 0 - 0}{3.5120} = 0.21865.$$

$$i=4 \Rightarrow D_4 = A_{44} - L_{41}^2 D_1 - L_{42}^2 D_2 - L_{43}^2 D_3$$

$$= 3.1520 - 0 - (-0.670219)^2 (2.9841) - (-0.21865)^2 (3.5120) = 1.64366$$

$$\begin{aligned} j=5 \Rightarrow L_{54} &= \frac{A_{54} - L_{51} D_1 L_{41} - L_{52} D_2 L_{42} - L_{53} D_3 L_{43}}{D_4} \\ &= \frac{-1.1520 - 0 - 0 - (0.21865)(3.5120)(-0.21865)}{1.64366} = -0.598724 \end{aligned}$$

$$i=5 \Rightarrow D_5 = A_{55} - L_{51}^2 D_1 - L_{52}^2 D_2 - L_{53}^2 D_3 - L_{54}^2 D_4$$

$$= 3.1520 - 0 - 0 - (0.21865)^2 (3.5120) - (-0.598724)^2 (1.64366) = 2.3949$$

Forward Substitution ($\mathbf{Ly} = \mathbf{b}$)

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0.21865 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & -0.670219 & -0.21865 & 1 & 0 \\ 0 & 0 & 0.21865 & -0.598724 & 1 \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ -0.04 \\ 0 \end{Bmatrix}$$

$$i=1 \Rightarrow y_1 = b_1 = 0$$

$$i=2 \Rightarrow y_2 = b_2 - L_{21}y_1 = 0$$

$$i=3 \Rightarrow y_3 = b_3 - L_{31}y_1 - L_{32}y_2 = 0$$

$$i=4 \Rightarrow y_4 = b_4 - L_{41}y_1 - L_{42}y_2 - L_{43}y_3 = -0.04$$

$$i=5 \Rightarrow y_5 = b_5 - L_{51}y_1 - L_{52}y_2 - L_{53}y_3 - L_{54}y_4 = 0 - (-0.598724)(-0.04) = -0.023949$$

Backward Substitution ($\mathbf{DL}^T \mathbf{x} = \mathbf{y}$)

$$\begin{bmatrix} 3.5120 & 0.21865 & 0 & 0 & 0 \\ 0 & 2.9841 & 0 & -0.670219 & 0 \\ 0 & 0 & 3.5120 & -0.21865 & 0.21865 \\ 0 & 0 & 0 & 1.64366 & -0.598724 \\ 0 & 0 & 0 & 0 & 2.3949 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ -0.04 \\ -0.023949 \end{Bmatrix}$$

$$i=5 \Rightarrow x_5 = \frac{-0.023949}{2.3949} = -0.01$$

$$i=4 \Rightarrow x_4 = \frac{y_4}{D_4} - L_{45}x_5 = \frac{-0.04}{1.64366} - (-0.598724)(-0.01) = -0.0303232$$

$$i=3 \Rightarrow x_3 = \frac{y_3}{D_3} - L_{34}x_4 - L_{35}x_5$$

$$= \frac{0}{3.5120} - (-0.21865)(-0.0303232) - (0.21865)(-0.01) = -0.00444367$$

$$i=2 \Rightarrow x_2 = \frac{y_2}{D_2} - L_{23}x_3 - L_{24}x_4 - L_{25}x_5$$

$$= \frac{0}{2.9841} - 0 - (-0.670219)(-0.0303232) - 0 = -0.0203232$$

$$i=1 \Rightarrow x_1 = \frac{y_1}{D_1} - L_{12}y_2 - L_{13}y_3 - L_{14}y_4 - L_{15}y_5$$

$$= \frac{0}{3.5120} - (0.21865)(-0.0203232) - 0 - 0 - 0 = 0.00444367$$

10.2.2 SPECIAL CASES

Sometimes solution to $\mathbf{Ax} = \mathbf{b}$ is required with additional conditions of the following forms.

(a) $x_i = c$

(b) $c_i x_i + c_j x_j = c$

We will handle each form separately only because of efficiency and accuracy considerations.

Case (a)

Consider the set of three equations

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ b_3 \end{Bmatrix} \quad (10.2.35a)$$

with the additional condition as

$$x_2 = c \quad (10.2.35b)$$

where c is a constant.

We will rewrite Eqn. (10.2.35a) as three separate equations

$$A_{11}x_1 + A_{12}x_2 + A_{13}x_3 = b_1 \quad (10.2.36a)$$

$$A_{21}x_1 + A_{22}x_2 + A_{23}x_3 = b_2 \quad (10.2.36b)$$

$$A_{31}x_1 + A_{32}x_2 + A_{33}x_3 = b_3 \quad (10.2.36c)$$

Utilizing the condition $x_2 = c$ in the above equations, we have

$$A_{11}x_1 + (0)x_2 + A_{13}x_3 = b_1 - A_{12}c \quad (10.2.37a)$$

$$(0)x_1 + (1)x_2 + (0)x_3 = c \quad (10.2.37b)$$

$$A_{31}x_1 + (0)x_2 + A_{33}x_3 = b_3 - A_{32}c \quad (10.2.37c)$$

Note carefully that Eqns. (10.2.36a) and (10.2.37a) are the same if $x_2 = c$. We have merely taken the x_2 term to the right-hand side where it belongs since x_2 is strictly no longer an unknown. Similar comments are valid for Eqns. (10.2.36c) and (10.2.37c). In order that (a) we do not change the number of equations, and (b) recognize that $x_2 = c$, suitable changes have been made to Eqn. (10.2.36b) and Eqn. (10.2.37b) is merely $x_2 = c$. We can rewrite Eqns. (10.2.37) as

$$\begin{bmatrix} A_{11} & 0 & A_{13} \\ 0 & 1 & 0 \\ A_{31} & 0 & A_{33} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} b_1 - A_{12}c \\ c \\ b_3 - A_{32}c \end{Bmatrix} \quad (10.2.38)$$

This approach is known as the Elimination Approach. The advantage of the equations in the above form is that the number of equations remains the same and that the coefficient matrix is still square (and symmetric if the original \mathbf{A} was symmetric). From a viewpoint of implementing the solution procedure in the form of a computer program, these are desirable properties.

General Procedure: If condition $x_j = c$ is to be imposed, implement the following three steps.

Step 1: Modify the right-hand side vector as $b_i = b_i - A_{ij}c$, $i = 1, \dots, n$.

Step 2: Modify the coefficient matrix as

$$A_{ij} = 0, i = 1, \dots, n$$

$$A_{ji} = 0, i = 1, \dots, n$$

Step 3: Set $A_{jj} = 1$.

This three-step process must be applied for each condition $x_j = c$.

Case (b)

We will solve this case using the Penalty Approach. We will restrict our attention to the case where \mathbf{A} is symmetric and positive definite. From calculus, we have the condition that the minimum of

$$\Pi(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} - \mathbf{x}^T \mathbf{b} + \frac{1}{2} C (c_i x_i - c_j x_j - c)^2 \quad (10.2.39)$$

where C is a large number, is the solution to the original constrained problem. Note that Π takes on a minimum value when $c_i D_i - c_j D_j - c$ is zero (or, numerically very small). Minimizing Π or computing $\partial \Pi / \partial \mathbf{x} = 0$ yields the following equations

$$\begin{bmatrix} A_{11} & A_{1i} & A_{1j} & A_{1n} \\ .. & .. & .. & .. \\ A_{i1} & A_{ii} + Cc_i^2 & A_{ij} + Cc_i c_j & A_{in} \\ .. & .. & .. & .. \\ A_{j1} & A_{ji} + Cc_i c_j & A_{jj} + Cc_j^2 & A_{jn} \\ .. & .. & .. & .. \\ A_{n1} & A_{ni} & A_{nj} & A_{nn} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_i \\ x_j \\ x_n \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_i + Ccc_i \\ b_j + Ccc_j \\ .. \\ b_n \end{Bmatrix} \quad (10.2.40)$$

Note that the modified equations $\mathbf{Ax} = \mathbf{b}$ still has a symmetric and positive definite coefficient matrix. The only question left to answer is what is the suitable value for the large number C . A popular choice that seems to work effectively, is to make the constant a function of the largest element in the coefficient matrix.

$$C = 10^4 \max |A_{pq}|, 1 \leq p, q \leq n \quad (10.2.41)$$

Example 10.2.5

Modify the following equations

$$\begin{bmatrix} 10 & -5 & 2 \\ 3 & 20 & 5 \\ -2 & 7 & 15 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 6 \\ 58 \\ 57 \end{Bmatrix}$$

so that the condition $x_2 = 3$ can be imposed.

Solution

Using the general procedure listed above, we have the following modified equations.

$$\begin{bmatrix} 10 & 0 & 2 \\ 0 & 1 & 0 \\ -2 & 0 & 15 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 6 + 5(3) \\ 3 \\ 57 - 7(3) \end{Bmatrix} = \begin{Bmatrix} 21 \\ 3 \\ 36 \end{Bmatrix}$$

These equations can now be solved using LU Decomposition.

Example 10.2.6

Modify the following equations

$$\begin{bmatrix} 10 & -5 & 2 \\ -5 & 20 & 5 \\ 2 & 5 & 15 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 6 \\ 58 \\ 57 \end{Bmatrix}$$

so that the condition $2x_1 + x_3 = 3$ can be imposed.

Solution

Using the procedure listed above, we have the following modified equations.

$$\begin{bmatrix} 10 + 20(10^4)2^2 & -5 & 2 + 20(10^4)(2)(1) \\ -5 & 20 & 5 \\ 2 + 20(10^4)(2)(1) & 5 & 15 + 20(10^4)1^2 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 6 + 20(10^4)(3)(2) \\ 58 \\ 57 + 20(10^4)(3)(1) \end{Bmatrix}$$

Or, simplifying

$$\begin{bmatrix} 800010 & -5 & 400002 \\ -5 & 20 & 5 \\ 400002 & 5 & 200015 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 1200006 \\ 58 \\ 600057 \end{Bmatrix}$$

These equations can now be solved using LDL^T Decomposition.

10.2.3 ITERATIVE SOLVERS

TBC

10.3 Case Study: A Matrix ToolBox

We saw the development of the `CVector` and `CMatrix` classes in Chapter 9. In this section, we will see how to leverage that development with building a matrix toolbox that will be useful in obtaining numerical solutions to engineering problems.

Example Program 10.3.1 Matrix Toolbox

We will now develop the function prototypes for the functions in the matrix toolbox. The template class is called `CMatToolBox`. In the table below, we present the details of these template functions that use `CVector` and `CMatrix` classes.

Function Prototype	Remarks
<code>bool Add (const CVector<T>& A, const CVector<T>& B, CVector<T>& C);</code>	Computes $\mathbf{c} = \mathbf{a} + \mathbf{b}$
<code>bool Subtract (const CVector<T>& A, const CVector<T>& B, CVector<T>& C);</code>	Computes $\mathbf{c} = \mathbf{a} - \mathbf{b}$
<code>bool DotProduct (const CVector<T>& A, const CVector<T>& B, T& c);</code>	Computes $c = \mathbf{a} \cdot \mathbf{b}$
<code>bool Normalize (CVector<T>& A);</code>	Computes $\frac{\mathbf{a}}{\ \mathbf{a}\ }$
<code>bool Scale (CVector<T>& A, T c);</code>	Computes $\mathbf{a} = c\mathbf{a}$ where c is a constant.
<code>T MaxValue (const CVector<T>& A);</code>	Computes $\max\{a_i, i = 1, 2, \dots, n\}$
<code>T MinValue (const CVector<T>& A);</code>	Computes $\min\{a_i, i = 1, 2, \dots, n\}$
<code>T TwoNorm (const CVector<T>& A);</code>	Computes $\ \mathbf{a}\ $
<code>T MaxNorm (const CVector<T>& A);</code>	Computes $\ \mathbf{a}\ _\infty$
<code>bool CrossProduct (const CVector<T>& A, const CVector<T>& B, CVector<T>& C);</code>	Computes $\mathbf{c} = \mathbf{a} \times \mathbf{b}$
<code>bool Add (const CMatrix<T>& A, const CMatrix<T>& B, CMatrix<T>& C);</code>	Computes $\mathbf{C} = \mathbf{A} + \mathbf{B}$
<code>bool Subtract (const CMatrix<T>& A, const CMatrix<T>& B, CMatrix<T>& C);</code>	Computes $\mathbf{C} = \mathbf{A} - \mathbf{B}$
<code>bool Multiply (const CMatrix<T>& A, const</code>	Computes $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$

CMatrix<T>& B, CMatrix<T>& C);	
bool Determinant (const CMatrix<T>& A, T& c);	Computes $c = \det(\mathbf{A})$
bool Scale (CMatrix<T>& A, T c);	Computes $\mathbf{A} = c(\mathbf{A})$ where c is a constant.
T MaxNorm (const CMatrix<T>& A);	Computes $\ \mathbf{A}\ _{\max}$
bool Transpose (const CMatrix<T>& A, CMatrix<T>& B);	Computes $\mathbf{B} = \mathbf{A}^T$
bool MatMultVec (const CMatrix<T>& A, const CVector<T>& x, CVector<T>& b);	Computes $\mathbf{b} = \mathbf{Ax}$
bool LUFactorization (CMatrix<T>& A, T TOL);	Computes $\mathbf{A} = \mathbf{LU}$. Overwrite \mathbf{A} with \mathbf{L} and \mathbf{U} .
bool LUSolve (const CMatrix<T>& A, CVector<T>& x, const CVector<T>& b);	Computes $\mathbf{L}\mathbf{Ux} = \mathbf{b}$. \mathbf{A} contains \mathbf{L} and \mathbf{U} .
bool AxEqb (CMatrix<T>& A, CVector<T>& x, CVector<T>& b, T TOL);	Computes $\mathbf{Ax} = \mathbf{b}$ using Gaussian Elimination.
bool LDLTFactorization (CMatrix<T>& A, T TOL);	Computes $\mathbf{A} = \mathbf{LDL}^T$. Overwrite \mathbf{A} with \mathbf{L} and \mathbf{D} . \mathbf{A} is a symmetric matrix.
bool LDLTSolve (const CMatrix<T>& A, CVector<T>& x, const CVector<T>& b);	Computes $\mathbf{LDL}^T \mathbf{x} = \mathbf{b}$. \mathbf{A} contains \mathbf{L} and \mathbf{D} .

A sample client code that uses four of the functions from the matrix toolbox is shown below.

main.cpp

```

1 #include <iostream>
2 #include "MatToolBox.h"
3
4 int main ()
5 {
6     const int ROWS=2, COLUMNS=3;
7     CMatToolBox<float> MTB;
8     CVector<float> fVA(COLUMNS), fVB(COLUMNS),
9     fVC(COLUMNS);
10
11    for (int i=1; i <= COLUMNS; i++)
12    {
13        fVA(i) = static_cast<float>(i);

```

```
14         fVB(i) = static_cast<float>(i*i);
15     }
16
17     // vector addition
18     if (MTB.Add (fVA, fVB, fVC))
19     {
20         MTB.Display ("Vector A", fVA);
21         MTB.Display ("Vector B", fVB);
22         MTB.Display ("Vector C = A + B", fVC);
23     }
24     else
25         std::cout << "Vector Add error.\n";
26
27     // matrix-vector multiplication
28     CMatrix<float> fMA(ROWS,COLUMNS);
29     for (int i=1; i <= ROWS; i++)
30     {
31         for (int j=1; j <= COLUMNS; j++)
32         {
33             fMA(i,j) = static_cast<float>(i+j);
34         }
35     }
36     fVC.SetSize (ROWS);
37     if (MTB.MatMultVec(fMA, fVB, fVC))
38     {
39         MTB.Display ("Matrix A", fMA);
40         MTB.Display ("Vector B", fVB);
41         MTB.Display ("Vector C = A*B", fVC);
42     }
43     else
44         std::cout << "MatMultvec error.\n";
45
46     return 0;
47 }
```

Summary

Matrix algebra forms the foundation of most of numerical engineering analysis. We saw a good number of matrix operations and solution techniques in this chapter. We will see more solution techniques such as the numerical solution to eigenproblems etc. in later chapters.

EXERCISES

Appetizers*Problem 10.1*

Solve the following set of equations by hand using LU Factorization. Also compute the residual vector and the absolute and relative errors.

$$\begin{bmatrix} 6 & 2 & 2 \\ 2 & 2/3 & 1/3 \\ 1 & 2 & -1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} -2 \\ 1 \\ 0 \end{Bmatrix}$$

Main Course*Problem 10.2*

Solve the following set of equations by hand using Cholesky Decomposition. Also compute the residual vector and the absolute and relative errors.

$$\begin{bmatrix} 2.25 & -3.0 & 4.5 \\ -3.0 & 5.0 & -10.0 \\ 4.5 & -10.0 & 34.0 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 1000.0 \\ 0 \\ -500.0 \end{Bmatrix}$$

Problem 10.3

Write two functions that help solve $\mathbf{Ax} = \mathbf{b}$ where \mathbf{A} is a symmetric, positive definite matrix stored in the rectangular format. The function prototypes are given below.

```
bool CholeskyFactorizationBanded (CMatrix<T>& A, T TOL);
bool CholeskySolveBanded (const CMatrix<T>& A, CVector<T>& x);
```

Problem 10.4

Write two functions that help solve $\mathbf{Ax} = \mathbf{b}$ where \mathbf{A} is a symmetric, positive definite matrix stored in the skyline format. The function prototypes are given below.

```
bool LDLTFactorizationSkyline (CVector<T>& A, CVector<int>& DLoc,
                                int n, T TOL);
bool LDLTSolveSkyline (const CVector<T>& A, const CVector<int>&
                           DLoc, CVector<T>& x, int n);
```

Numerical Analysis Concepts

Problem 10.5

Complete the implementation of the matrix toolbox discussed in Example Program 10.3.1. Write a client code that will properly test the functions in the toolbox. Here are the programming details to note and follow.

- (1) Functions should carry out the basic checks to see if the arrays are of the proper size. Functions should return a false value if they detect an input error.
- (2) Do not use `std::cout` or `std::cin` statements in these functions unless specifically required. For example, the `Display` function should use `std::cout`.
- (3) Do not declare temporary or local arrays in any of these functions except the function that computes the determinant.
- (4) For both **LU** and **LDLT^T** factorization approaches, assume that the `LUFactorization` and the `LDLTFactorization` functions are called first (just once) and that the `LUSolve` and `LDLTSolve` functions can then be called with the factorized **A** matrix as many times as needed.

Regression Analysis

"The person who knows "how" will always have a job. The person who knows "why" will always be his boss." Diane Ravitch

'Real learning comes about when the competitive spirit has ceased.' J. Krishnamurti

In earlier courses in Statics and Deformable Solids (or, Strength of Materials), most of the structural systems were statically determinate.

Objectives

- To understand the syntax of C++ programs.
- To understand the concepts associated with data types, variables, arithmetic expressions, assignment statements and simple input and output.
- To understand and practice writing C++ programs.

11.1 Interpolation and Polynomial Approximation

11.2 Curve Fitting

EXERCISES

Appetizers

Main Course

Numerical Analysis Concepts

File Handling

"The person who knows "how" will always have a job. The person who knows "why" will always be his boss." Diane Ravitch

'Real learning comes about when the competitive spirit has ceased.' J. Krishnamurti

Almost all computer programs require some input from the user and they generate some output. In programs that are graphical-user interfaced (GUI), the input is via the keyboard, mouse or even an external file. The output is shown graphically or as a report either on the screen, or in the printed form, or stored in an external file such that it can be viewed later. There are innumerable occasions when data need to be transported from one program to another or data created by a program need to be retained even after the program has finished execution. Typically, the data under these scenarios, are stored in an external file on a computer's hard disk. If the contents of the file can be viewed meaningfully by a text editor¹ then the files have text data in them. Otherwise, the files are holding binary data. In this chapter, we will see how to handle files and manipulate data in them using C++.

Some of the C++ file handling concepts requires an understanding of advanced object-oriented concepts such as inheritance. However, the value of file handling takes precedence at this stage. We will see the advanced OO concepts in Chapter 13 when we will be in a better position to understand all the nuances of the C++ file handling classes.

Objectives

- To understand what are external files.
- To understand how to read the data from an input file.
- To understand how to write data to an output file.

¹ Wordpad© or Notepad© are examples of Windows text editors. vi or emacs are examples of Linux text editors.

12.1 File Streams

Input and output in C++ are byte-oriented. In other words, a sequence of bytes is read from or written to *streams*. Fig. 12.1.1 shows the C++ class hierarchy. One can look at class hierarchy as building blocks. The base class is at the foundation level and other classes (derived classes) have functionalities that are built on top of the base class functionalities. We already have seen the basic IO class – `iostream`. This is the class that has been used in most of the prior examples to ‘read the input’ via `cin` and ‘write the output’ via `cout`.

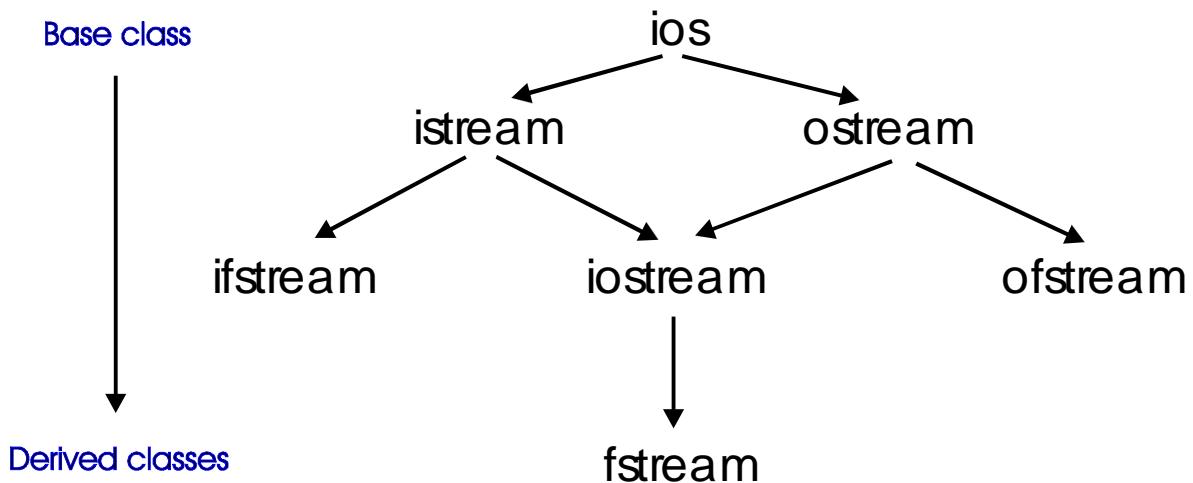


Fig. 12.1.1 Class hierarchy

In this section, we will see how to read data from an external file and how to output information into an external file. Note that files are either text (meaning that they can be viewed in a text editor such as Windows Notepad etc.) or binary (meaning that the information is stored as 0’s and 1’s whose meaning can be interpreted by the program reading and writing the data). Text files can usually be accessed sequentially. Random access can take place with binary files.

12.2 File Input and Output

12.2.1 Opening a text file for reading

A file must be opened first before data can be read from it (or written to it). To open a file, one must know the name of the file. In other words, a file can be opened for reading provided it is an existing file. Typically, one creates this file using a text editor.

To open a file for reading, we need to include the appropriate C++ header file. The header file and the accompanying statements are as follows.

```
#include <fstream>
using std::ifstream;
```

First we need to create an object associated with the `ifstream` class. The general syntax is the usual syntax associate with the declaration of a class-related object.

```
ifstream object_name;
```

For example,

```
ifstream FileForInput;
```

`FileForInput` is the variable or object. Once the object is declared, it can then be used to open the file using the `open` member function. For example,

```
FileForInput.open ("DataSet1.dat", ios::in);
```

opens a file called `DataSet1.dat` for reading only. The member function `open` has two parameters. The first is the character string that contains the name of the file (e.g. `DataSet1.dat`) and the second defines the `openmode` value. The `openmode` value in this case is `ios::in` that indicates that the file exists and is being opened for reading only. Once the file is opened, the references to the file do NOT take place with respect to the file name. Instead, the object name is used for the rest of the program as an alias for the file name.

When the file is no longer needed, it must be closed before the program terminates using the `close` member function. For example,

```
FileForInput.close ();
```

closes the file that was opened earlier.

To read the data from the file, we can use the `>>` operator. For example, if we wish to read an integer followed by a float from the file, using the example discussed above, we would have the following statement.

```
FileForInput >> nIntV >> fFloatV;
```

where `nIntV` is an `int` variable and `fFloatV` is a `float` variable.

12.2.2 Opening a text file for writing

A file must be opened first before data can be written to it. To open a file, one must know the name of the file. A file can be opened for writing so that (a) it appends data to an existing file, (b) it overwrites an existing file, or (c) it creates a new file. The header file and the accompanying statements are as follows.

```
#include <fstream>
using std::ofstream;
```

First we need to create an object associated with the `ofstream` class. For example,

```
ofstream FileforOutput;
```

declares `FileForOutput` as an `ofstream` variable or object. Once the object is declared, it can then be used to open the file. Two examples are shown below.

```
FileforOutput.open ("DataSet1.out", ios::out);
FileforOutput.open ("DataSet1.out", ios::out | ios::app);
```

In the above examples, `ios::out` indicates that the file is created if it does not exist or if it exists, the contents are destroyed. Similarly, `ios::out | ios::app` indicates that if the file exists, then new information must be appended to the end of the file. When the file is no longer needed, it must be closed before the program terminates using the `close` member function. For example,

```
FileForOutput.close ();
```

closes the file. It is possible that you may not see the expected contents in a file if you do not close the file. This is because the contents are buffered. The contents are put in a special location and are written to the file only if the buffer is full or the `close` member function is invoked. The member function `flush` can be explicitly called to ‘flush the buffer’; `flush` is automatically called by the `close` function.

To write data to a file, we can use the `<<` operator. For example, if we wish to write an integer followed by a float on a single line, using the example discussed above, we would have the following statement.

```
FileForOutput << nIntV << " " << fFloatV << '\n';
```

The following table summarizes the file `openmode` flags.

Flag	Remarks
<code>in</code>	Opens the file for reading (default action for <code>ifstream</code>)
<code>out</code>	Opens a file for writing (default action for <code>ofstream</code>)
<code>app</code>	Appends (at the end) when writing
<code>ate</code>	Positions at the end of the file after opening. <code>ate</code> is the short form for at end.
<code>trunc</code>	Truncates the contents of the file by removing the current contents
<code>binary</code>	For binary (non-text) files.

Tip: Text files are created and accessed sequentially. If a file is being created, the contents are first written to line 1, followed by line 2, and so on. A new line is created only if the newline character is output to the file. One cannot go back and rewrite a specific line in a text file. Similarly, if a file is being read, the contents are read sequentially starting at the beginning of the file. To read a specific line that is not the first line in the file, one must skip the previous lines by reading the contents and discarding them, before reading the required line.

12.2.3 Stream States

Streams have an associated state that identifies if the IO was successful or not, and possibly the reason for the failure. One must safeguard against error conditions when writing a program. C++ provides several member functions that can be used for error handling by accessing the state-bit value.

Constant	Remarks
<code>badbit</code>	Fatal error with undefined state.
<code>eofbit</code>	End-of-file was encountered.
<code>failbit</code>	An IO operation was unsuccessful.
<code>goodbit</code>	Everything is OK. Other bits are not set.

fail: This member function is used to test if the stream operation has failed or not by accessing the `failbit` or `badbit` value. The `fail` function returns `true` if the last stream operation was unsuccessful. For example, to see if the `open` function worked or not, we could have the following code.

```
FileForInput.open ("DataSet1.dat", ios::in);
if (FileForInput.fail())
{
    std::cout << "Could not open specified output file.\n";
    // take appropriate action
    ...
}
```

eof: This member function is used to test and see if the end-of-file condition has been reached by accessing the `eofbit` value. A file cannot be read beyond its end. Here is an example code that repeatedly reads an integer from a file until there is no more input.

```
int nv;
while (!FileForInput.eof())
{
    FileForInput >> nv;
    // do what's appropriate with the input
    ...
}
```

Note that the end-of-file character is a special character that is not visible on most text editors. The computer's file system automatically adds this marker to every file.

clear: This member function clears all of the standard input status flags. For example, to be able to read a file once again after the end-of-file is reached, one can use the code shown below.

```
FileForInput.clear();
```

The overloaded form of this function is **clear (state)** that clears all and sets the **state** flag.

good: This member function checks to see if the stream is OK by accessing the **goodbit** value.

bad: This member function checks to see if a fatal error has occurred by accessing the **badbit** value.

rdstate: This member function returns the currently set flags. The following example checks to see if the **failbit** is set and clears it if necessary.

```
if (FileForInput.failbit())
{
    FileForInput.clear (FileForInput.rdstate() & ~std::ios::failbit);
}
```

Example Program 12.2.1 File Handling

Next we will see a sample program that illustrates how to read from an input file and how to write a tabular output to an output file.

Problem Statement: Write a program to read an input file. Assume that the file has an unknown number of lines of input. However, each line has an integer and 3 real numbers. The integer represents the point number while the real numbers represent the (x, y) coordinates of and the temperature at that point. While reading the file, create a properly formatted file that has the same information in an easy to read tabular form. Use the sample data file given below to test the program.

```
1 12.0 -45.6 33.3
2 16.0 45.6 88.6
3 -3.3 -4.4 1.1
```

The source code for the developed program is given below.

main.cpp

```
1 #include <string>
2 #include <iostream>
3 #include <iomanip>
4 #include <fstream>
5
6 int main ()
7 {
```

```

8      // filename to be stored here
9      std::string szInputFileName, szOutputFileName;
10
11     // input file
12     std::ifstream FileInputStream;
13
14     // output file
15     std::ofstream FileOutputStream;
16
17     // get the input file name from the user
18     int nSuccess = 0;
19     do
20     {
21         std::cout << "Input file name (including extension)? ";
22         std::cin >> szInputFileName;
23         std::cout << "Trying to read from file " << szInputFileName
24             << std::endl;
25
26         // open the file
27         // ios::in indicates the file is to be opened for
28         // reading only
29         FileInputStream.open (szInputFileName.c_str(), std::ios::in);
30
31         if (FileInputStream.fail())
32         {
33             std::cout << "Unable to open input file" << std::endl;
34             // must set the goodbit so that the open statement executes
35             FileInputStream.clear ();
36         }
37         else
38             nSuccess=1; // file opened successfully
39     } while (nSuccess == 0);
40
41     // get the output file name from the user
42     nSuccess = 0;
43     do
44     {
45         std::cout << "Output file name (including extension)? ";
46         std::cin >> szOutputFileName;
47         std::cout << "Creating file " << szOutputFileName << std::endl;
48
49         // open the file
50         // ios::out indicates the file is to be opened for
51         // writing only
52         FileOutputStream.open (szOutputFileName.c_str(), std::ios::out);
53         if (FileOutputStream.fail())
54         {
55             std::cout << "Unable to open output file" << std::endl;
56             FileOutputStream.clear ();
57         }
58         else
59             nSuccess=1; // file opened successfully
60     } while (nSuccess == 0);
61
62     // read the input and write the output
63     int nPointNumber;
64     float fx, fy, fTemp;

```

```

65
66     // set column headers for output file
67     FileOutput << setiosflags(std::ios::left) << std::setw (7)
68         << "PID" << std::setw (15) << "X" << std::setw (15)
69         << "Y" << std::setw (15) << "Temp" << std::endl;
70     FileOutput << setiosflags(std::ios::left) << std::setw (7)
71         << "---" << std::setw (15) << "-" << std::setw (15)
72         << "-" << std::setw (15) << "----" << std::endl;
73
74     int nLines = 0;
75     for (;;)
76     {
77         FileInputStream >> nPointNumber;
78         if (FileInputStream.eof()) break; // end of file reached?
79         FileInputStream >> fX >> fY >> fTemp;
80         if (FileInputStream.eof()) break; // end of file reached?
81         nLines++;
82         FileOutput << setiosflags(std::ios::left)
83             << std::setw(6) << nPointNumber
84             << std::setw(15) << fX
85             << std::setw(15) << fY
86             << std::setw(15) << fTemp << "\n";
87     }
88
89     // close the files and terminate the program
90     std::cout << szInputFileName << " was read and had " << nLines
91         << " lines of input.\n";
92     FileInputStream.close ();
93     FileOutput.close ();
94
95     // all done
96     return (0);
97 }
```

In line 4 the appropriate header file `<fstream>` is included. For opening both the input and out files, a `do` loop is used that repeatedly executes until the file is successfully open. Note the way the input file name obtained from the user is passed to the `open` function (lines 29 and 52). The `open` function does not permit a `std::string` object; instead it expects to see a `char` string. The `std::string` member function `c_str()` provides the address of the location where the string is stored. Also note that the `clear` member function is called if for some reason the file cannot be opened.

The reading of the input file starts on line 75. Each input line is read in two parts. First the point number is read (line 77) followed by the statement to read the coordinates and the temperature on line 79. After both reads, the end-of-file condition is checked using the `eof` member function. The data is immediately written to the output file (line 82). The formatting statements (using `setiosflags` and `std::setw` functions) are similar to those that we saw before. They ensure that the column headings and the column data are properly aligned. Finally, the `close` member functions are called to close both the input and output files (lines 92-93).

Tip: The `>>` operator will read past leading and trailing blanks as well as newline character to get to the next input. In other words, one can reform the sample data file for the program as follows.

```

1
12.0
-45.6
33.3
2 16.0 45.6 88.6
3 -3.3 -4.4 1.1

```

12.3 Advanced Usage

C++ provides a very rich set of functionalities for file manipulation. We discuss some of the more useful features here.

Reading a single character

A single character can be read by using the `get` member function whose prototype is shown below.

```
istream& istream::get (char& c);
```

This function assigns the next character to the passed argument and returns the stream. The state of the stream indicates if the read was successful. Here is an example.

```

char c;
FileForInput.get(c);

```

Reading an entire line

The contents of an entire line can be read using the `getline` member functions.

```
istream& istream::getline (char* str, streamsize count);
istream& istream::getline (char* str, streamsize count, char delim);
```

This function reads up to `count-1` characters into `str`. However, the reading is terminated if the next character to be read is the newline character (`\n`) that is the default or the `delim` character. The terminating character is not read. The state of the stream indicates if the read was successful. Here is an example.

```

const int MAXCHARS = 256;
char szUserInput[MAXCHARS];
FileForInput.getline(szUserInput, MAXCHARS);

```

Passing a `fstream` object to a function

A file, once opened, can be used in any part of the program where the file stream object is visible. Sometimes, it may be desirable to pass the file stream object to a function. Here is an example function prototype.

```
void StoreMessage (fstream& OutputFile, const std::string& szMessage);
```

As we have seen before with other objects, stream objects should be passed as references.

Example Program 12.3.1 Advanced File Handling

In this example we will see how to parse and read an input file.

Problem Statement: Write a program to read an input file called **dbfile.dat** that has travel-related data for a trucking company. A sample file is shown below.

```
Rem: File generated on 15-Jan-2005
Phoenix 12-Jan-2005 13500
San_Diego 12-Jan-2005 13945
Sacramento 14-Jan-2005 15456
Rem: File generated on 18-Jan-2005
Portland 16-Jan-2005 17800
Seattle 17-Jan-2005 18400
```

```
Rem: File generated on 25-Jan-2005
Eugene 20-Jan-2005 19500
San_Jose 21-Jan-2005 20600
Las_Vegas 23-Jan-2005 22300
Phoenix 24-Jan-2005 23100
```

A typical line in the file is either a line that starts with **Rem:** or is a blank line or is a data line. The data line contains three fields – the city name, the date of travel and the current odometer reading.

main.cpp

```
1 #include <iostream>
2 #include <fstream>
3 #include <sstream>
4
5 bool ReadNextLine (std::istream& FileInputStream,
6                     char *szInputString, const int MAXCHARS)
7 {
8
9     // read the line
10    FileInputStream.getline (szInputString, MAXCHARS);
11
12    // end-of-file or fatal error returns false
13    if (FileInputStream.eof())
14        return false;
15    if (FileInputStream.fail())
16        FileInputStream.clear (FileInputStream.rdstate() & ~std::ios::failbit);
17    if (FileInputStream.bad())
18        return false;
19
20    // successful read
21    return true;
22 }
23
```

```

24 int main ()
25 {
26     // open the db file
27     std::ifstream FileInput;
28     FileInput.open ("dbfile.txt", std::ios::in);
29     if (!FileInput)
30     {
31         std::cout << "Unable to open dbfile.txt.\n";
32         return 1;
33     }
34
35     // read the contents of the file
36     const int MAXCHARS = 256;
37     char szInputString[MAXCHARS];
38     std::string szTempString;
39     int nLine = 0;
40     std::string szCity, szDate;
41     int nMileage;
42
43     // now read the file contents till eof or error occurs
44     while (ReadNextLine (FileInput, szInputString, MAXCHARS))
45     {
46         std::cout << ++nLine << ":" << szInputString << '\n';
47         szTempString = szInputString;
48         // if line is not a remark or if string is not empty,
49         // read the data on that line
50         if (szTempString.substr(0,4) != "Rem:" &&
51             szTempString.length() > 0)
52         {
53             std::istringstream szFormatString (szTempString);
54             szFormatString >> szCity >> szDate >> nMileage;
55             std::cout << "City: " << szCity << ". Date: " << szDate
56                         << ". Mileage: " << nMileage << '\n';
57         }
58     }
59
60     return 0;
61 }
```

The fundamental idea in this approach is to read the entire line one at a time and then parse the input based on what we expect to see on a typical line. Reading a line is achieved in the function `ReadNextLine`. The input is read into character string `szInputString` using the `getline` function (line 10). Immediately error checks are carried out (lines 13-18). If an end-of-file condition is reached or if a fatal error is encountered, the function returns a `false` value that can then be used to terminate the reading of the file. Otherwise, a `true` value is returned.

The main program starts by opening the database file, `dbfile.txt`. The `ReadNextLine` function is called in a while loop until a false value is returned because of file read error. It is assumed that each line will not have more than 255 characters. Once the contents of a line are captured in the char string, the features of the `std::string` class are used to parse the line. First, in line 47, the contents are copied into a `std::string` variable, `szTempString`. Now we determine if the input line is a remark line or a blank line by checking to see if the first four characters are `Rem:` or if the length of the input string is zero, respectively. If both these conditions are not satisfied, then we can assume that the input

line contains data. The `istringstream` object is then used to read (parse) the three input fields. First, in line 53, the contents are copied into the `istringstream` object. Next, in line 54, the actual reading is done. Error checking is not carried out in the example code but one could call an appropriate member function (e.g. `fail()` or `bad()`) after line 54.

Binary File Access

As we saw with text files, access to the contents of the file is sequential in nature. However, there are applications where very fast access to the data is needed and the data need not exist in the text form. C++ provides functions that can be used to read and write data at specific locations in a file with the read and write operations taking place in a random order. We will now see how this can be carried out.

First we need to open the file (a) for both reading and writing, and (b) in a binary mode. This can be done using the `open` member function that we saw before. However, we need to specify additional parameters. For example

```
BinaryFile.open (szFileName, std::ios::binary | std::ios::out |
                 std::ios::in);
```

where `BinaryFile` is a `fstream` object. This form of the `open` statement opens the file for both reading and writing in a random manner. Few things to note about random access files.

- (1) Data cannot be read unless data is written first to the file.
- (2) The file needs to be positioned first before reading from or writing to that location. It is possible to overwrite the contents at the specified location.
- (3) The IO operations take place in **bytes** using `char` data type and type casting is done during the function calls.

The following functions are used for reading and writing. The letter **g** is used to signify get and the letter **p** is for put.

tellg: This function returns the read position.

seekg(pos): This function sets the read position as an absolute value.

seekg(offset,rpos): This function sets the read position as a relative value.

tellp: This function returns the write position.

seekp(pos): This function sets the write position as an absolute value.

seekp(offset,rpos): This function sets the write position as a relative value.

One can use an integral type to represent the file position. The constants `std::ios::beg`, `std::ios::cur`, and `std::ios::end` signify the beginning, current and end position of the file.

write(char*, n): This function can be used to write a scalar variable or a vector variable of size **n** bytes. The variable must be type cast as a `char *` before it can be written.

read(char*, n): This function can be used to read a scalar variable or a vector variable of size **n** bytes. The variable must be type cast as a `char*` before it can be read.

The example below shows how the type casting can be done.

Rewinding a file

A file can be rewound so that reading the file can once again start from the beginning of the file.

```
FileForInput.clear (std::ios_base::goodbit);
FileForInput.seekg (0L, std::ios::beg);
```

Example Program 12.3.2 Binary Files

In this example we will see how binary files can be used in a program.

Problem Statement: Write a program to read point-related data interactively, store (or write) the data in a binary file and read the data back to display on the screen.

We first slightly modify the `CPoint` class used in Example 9.1.1 by adding one more private member variable – `m_szTag` that will store the identification tag associated with the point. This variable is a `char` string variable designed to hold a maximum of 10 characters. The modified header and source files are not shown here. The binary file is implemented as a fixed-length record file. In other words, it is assumed that each point record has a constant size. If the size of one record is **n** bytes, then the first record starts at location 0, the second record at location **n-1**, and so on. The `sizeof` function can be used to obtain the size of a data type or a variable. For example

```
int nintSize = sizeof(int);
```

would return the size of an `int` for that particular computer system (typically 4 bytes). Similarly,

```
int nCPSize = sizeof(CPoint);
```

would return the size of the `CPoint` object. A diagram to help visualize a fixed length record file is shown in Fig. 12.3.1.

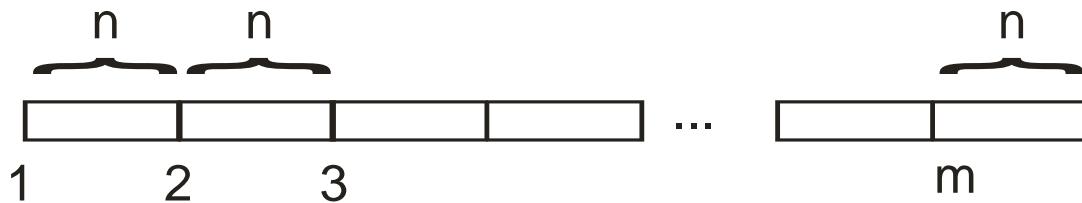


Fig. 12.3.1 Schematic diagram of a fixed length record file (record length is n , and there are m records currently in the file)

main.cpp

```

1  #include <cstdlib> // for exit()
2  #include <iostream> // for cin/cout
3  #include <fstream> // for db file
4  #include "point.h"
5
6  int main ()
7  {
8      std::fstream DBFile;
9      DBFile.open ("points.db", std::ios::binary | std::ios::out |
10                  std::ios::in    | std::ios::trunc);
11      if (!DBFile)
12      {
13          std::cout << "Unable to open database file.\n";
14          exit (0);
15      }
16      CPoint P;
17
18      // obtain data from user and create database
19      std::string szTag;
20      int nRecords = 0;
21      for (;;)
22      {
23          std::cout << "Input next data set (done tag terminates)\n";
24          std::cin >> P;
25          P.GetTag(szTag);
26          if (szTag == "done")
27              break;
28          ++nRecords;
29          DBFile.seekp ((nRecords-1)*sizeof(CPoint));
30          DBFile.write (reinterpret_cast<const char*>(&P),
31                        sizeof(CPoint));
32          if (DBFile.bad())
33          {
34              std::cout << "Fatal error during write.\n";
35              exit (0);
36          }
37      }
38
39      // read the database and display data
40      for (int i=1; i <= nRecords; i++)
41      {
42          DBFile.seekg ((i-1)*sizeof(CPoint));
43          DBFile.read (reinterpret_cast<char*>(&P),

```

```

44           sizeof(CPoint));
45     if (DBFile.bad())
46     {
47       std::cout << "Fatal error during read.\n";
48       exit (0);
49     }
50   P.Display ("Current point: ");
51 }
52
53 DBFile.close ();
54
55 return 0;
56 }
```

The appropriate header files are shown in lines 1-4. Files that are used for reading and writing must be associated with the `fstream` class. This declaration takes place on line 8 and the file is opened on line 9. Note the multiple `openmode` flags that are used. The `trunc` flag is needed if the file does not exist or if the file exists and the contents can be overwritten. If the file exists, then the `trunc` flag should be omitted. The loop starting at line 21 is used to interactively obtain the point data from the user. The `nRecords` variable keeps a track of how many point records have been created. It is then used to compute the location on the file where the reading or writing of the point data is to take place. The corresponding computations are on lines 29 and 42. In line 29, the `seekp` function is used to position the file before writing. Next the `write` function is used to write the data from the `CPoint` object (`P`) into the file. The `reinterpret_cast<const char*>` type casting is used to convert the data into `const char*` type before writing. Similarly, before reading the data back, the `seekg` function is used to position the file. The `read` member function is called to read the data. Once again type casting needs to take place and is done using the `reinterpret_cast<char*>` construct. Finally, the file is closed in line 53. One should note that C++ generates instructions to store the data associated with any object in contiguous memory locations. That is the reason why writing or reading `sizeof(CPoint)` bytes works properly and one does not have to write or read the tag, the x coordinate and the y coordinate values individually.

Tip: The above scheme needs to be used with care when writing/reading objects that have a pointer embedded in them and if the pointer is used to carry out resource management. In other words, not only the `sizeof` function may not return the correct length, but all the values may not be written to the disk. That is the reason why in this example, the point identification tag is stored as a `char` string not as a `std::string`. Different `std::string` objects may have different lengths. However, on the client side of the program, the `char` string implementation is hidden. Internally, in the `CPoint` class, the `strcpy` function is used to copy from the `std::string` variable to the `char` string variable. The `strcpy` function prototype is given below.

```
void strcpy (char * copyfrom, const char* copyto);
```

Summary

The ability to manipulate information stored in external files opens a lot of doors for us. Rarely do we have a single piece of software taking care of all our needs. Data need to be taken as input, manipulated using a numerically-based algorithm, and finally exported to use by other programs.

In this chapter we saw how files can be opened and closed, how text and binary data can be read from and written to files in a dynamic manner and how C++ provides functions to ensure that these are taking place in a safe and efficient manner. In later chapters, we will have several opportunities to use these concepts for solving practical problem.

EXERCISES

For the following problems, where appropriate use the **CVector** and **CMatrix** template classes to store the data. Design and implement appropriate classes to store and manipulate the data. In other words, you need to implement an object-oriented solution to each problem.

Appetizers

Problem 12.1

The program is required to create data to plot a cubic polynomial $y(x) = a + bx + cx^2 + dx^3$. The user interactively supplies the values of the 4 coefficients, the starting value of x , the ending value of x and the increment to be used. Write a program that will create an input file (say, comma separated file) for Microsoft Excel so that the graphing features in MS Excel can be used. Name this file **P12-1.csv**.

Main Course

Problem 12.2

Look at the statistical functions defined in *Problem 4.11*. Create a template class, **CStatPak**, that has the functionalities defined in the problem. Note that the class definition shown below is incomplete.

```
template <class T>
class CStatPak
{
public:
    T StatMean    (const CVector<T>& fV);
    T StatMedian  (const CVector<T>& fV);
    T StatStandardDeviation (const CVector<T>& fV);
    T StatVariance (const CVector<T>& fV);
    T StatCoVariance (const CVector<T>& fVA,
                      const CVector<T>& fVB);
private:
};
```

Now write a non-member function (that is called from the **main** ()) whose prototype is as follows.

```
int ReadInput (std::ifstream& InpFile, CVector<double>& dVStats);
```

The function should read the data from the file associated with **InpFile** and compute the statistical values using the **CStatPak** class. The results from the statistical analysis should be stored in the **dVStats** vector with the mean stored in the first location, followed by the median, standard deviation and finally variance. The return value is 0 if no error was encountered and 1 if an error was encountered. The input file has the following format.

Line 1: Number of data values, n

Line 2: First value

Line 3: Second Value

...

Line n+1: Last value

Carry out sufficient error checks to make this function robust. Create the template statistical functions in file `statpak.h` and the non-member functions in file `readinput.cpp`.

Problem 12.3

Write a program that will analyze experimental data and compare the results with an analytical model. The program will read data from an input file. It will then filter the data as per the specifications. It will compare the input to the theoretical value and then create the output file properly formatted. The sequence of events in the program is as follows.

1. Ask the user for the input file name. Open the input file.
2. Ask the user for the output file name. Open the output file.
3. Read the input file one line at a time.
4. Filter the data.
5. Create the output file.
6. Terminate the program.

Input File Format: The input file contains the following data in every line of the file obtained from an experiment.

The displacement value (at least one blank space) The load value

Both the displacement and load values are real numbers. An example input file (not real data):

```
0      10
0.0001 50
-0.1   200
0.001  1225
0.002  2500
0.004  5700
0.005  6400
0.007  8900
```

Filtering the Data: The displacement values (in inches) are expected to be between 0 and 10, and the load values (in pounds) are expected to be between 0 and 10000.

Output File Format: Create the output file that displays the data in tabular form as follows and using the filtered data, compute the area under the load-deflection curve.

Name:
Date:

Filtered Data

Data Point	Load (lb)	Experimental Displacement (in)	Theoretical Displacement (in)	Percent Difference

Bad Data

Data Point	Load (lb)	Displacement (in)

Area under the load-deflection curve = xxxx lb-in (Load on the y-axis and deflection on the x-axis).

Anaytical load-deflection function

The analytical function is as follows

$$\Delta = 10^{-10}(0.09P^3 - 3.9P^2 - 5.5P)$$

where P is the load in lb and Δ is the deflection in in .

Percent Difference

$$\% \text{ Difference} = \frac{\text{Analytical} - \text{Experimental}}{\text{Analytical}} \times 100$$

Problem 12.4

It is required to write a computer program to compute the amount of earthwork or excavation necessary at a site. The site (in plan view or x-y plane) is divided into a regular grid with the grid spacing at 3 feet in both directions. The contractor using surveying techniques has created an input file that contains the (x,y,z) values of these grid points.

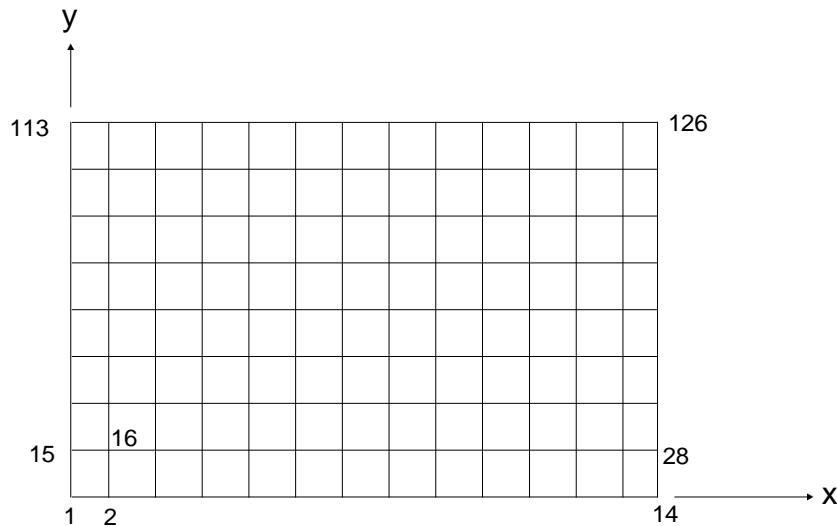


Fig. 1 Sample grid (all grid points are not labeled)

The program flow is expected to be as follows (you must use dynamically defined matrices to store the data).

Ask the user for the input file name. Open the input file.

Ask the user for the output file name. Open the output file.

Read the input file one line at a time and store the data.

Interpolate to fill missing values resulting from bad data (zero elevation). Interpolation is to seek the average value of all adjacent (good) grid points.

Adjacent points are the immediate next points along the row and column. Write a function to implement this task.

Compute the average elevation and excavation/earthwork quantity.

The volume of a grid of heights h_1, h_2, h_3, h_4 can be taken as $(\text{plan area})(h_{\text{avg}} - h_{\text{ex}})$ where h_{avg} is the average of the four values. Write a function to implement this task.

Create the output file. Write a function to implement this task.

Terminate the program.

Input File Format: The input file contains the following data on the first line.

Number of grid points in the x direction(blank)Number of grid points in the y direction

The second line contains the excavation elevation value.

FILE HANDLING

Z coordinate representing the excavation elevation (this is h_{ex})

Then it contains the (x,y,z) coordinates of the grid points rowwise.

X coordinate(blank)Y Coordinate(blank)Z Coordinate
X coordinate(blank)Y Coordinate(blank)Z Coordinate

Coordinate values are in feet.

An example of the input file:

```
5 4
8.1
0.0 0.0 12.1
3.0 0.0 12.5
6.0 0.0 11.9
9.0 0.0 11.5
12.0 0.0 11.4
0.0 3.0 12.6
3.0 3.0 12.4
6.0 3.0 12.3
9.0 3.0 11.9
12.0 3.0 0.0
0.0 6.0 12.3
3.0 6.0 12.1
6.0 6.0 12.0
9.0 6.0 0.0
12.0 6.0 12.0
0.0 9.0 12.2
3.0 9.0 12.3
6.0 9.0 12.1
9.0 9.0 11.9
12.0 9.0 11.8
```

The missing or bad data are highlighted.

Output File Format: Create the output file that displays the data as follows.

Name:

Date:

INPUT DATA

Grid Point	X Coordinate	Y Coordinate	Z Coordinate	Status
1				Good
2				Interpolated
3				Good

Average elevation of the site: xxx ft

Excavation elevation: xxx ft

EXCAVATION DATA	
Grid Points	Excavation (ft ³)
1-2-6-7	
Sum xxx ft ³	

C++ Concepts

Problem 12.5

Write a computer program to handle a cross-sections database. The database supports circular, hollow circular, rectangular and hollow rectangular cross-sections. Devise a scheme for identifying each cross-section in the database with a unique identification tag. The computed properties to be stored in the database include cross-sectional area and the two principle moments of inertia. Internally, the program should be able to store the data in a pre-defined set of units. However, it should allow the user to select the units at run time. The computer program should support the following top-level commands.

units: Used to specify the length units.

create: Used to create information in the database.

edit: Used to edit information already in the database.

find: Used to find the cross-sectional properties of a cross-section already in the database.

search: To find the closest cross-section that meets a specified search criterion.

delete: Delete the cross-section from the database.

Create and store the database file as a binary file whose life extends beyond the single execution of the program.

Problem 12.6

C++ does not have a function to check if a file exists. Develop a function with the following prototype.

```
bool FileExists (const char* filename);
```

that will return `true` if the file exists, `false` otherwise. Will this function work on all operating systems?

Chapter

13

Classes: Objects 303

“Education is a progressive discovery of our own ignorance.” Will Durant

*“When asked how much educated men were superior to those uneducated,” Aristotle answered,
“As much as the living are to the dead.”* Diogenes Laertius

We were introduced to fundamental object-oriented concepts in Chapter 7. We learnt about classes and objects. We learnt how to develop and write server code. We also learnt how to develop and write client code where objects would be declared and manipulated. As we saw in Chapters 8 and 9, object-oriented concepts help us define and use classes in a powerful yet safe manner. In this chapter we will first see the process that leads to the development of a computer program. Second, we will see the concepts associated with *inheritance*. What this means is that through the process of data abstraction, we will be able to enhance already available capabilities from existing classes by building specialized classes on top of them. Finally, we will see the concepts associated with *functors* and suggest how they can be used in writing user-defined functions.

Objectives

- To understand the basics of software engineering.
- To understand how to leverage already developed classes and build additional functionality.
- To understand the concepts of polymorphism.
- To understand the concepts associated with functors and use them as a tool in writing user-defined functions.

13.1 Software Engineering

So far the programs that we have developed have been comparatively small and easy to manage. The process followed in the development of the programs was more intuitive than formal. In this section, we will formalize the process - the development of a complete program should be a well-thought out process.

Software engineering is the development and study of the principles that enable any organization to predict and control quality, schedule, cost, cycle time, and productivity when acquiring, building, or enhancing software systems¹. Why is software engineering important? It is important because computer software, like any other product, is useful and expensive to develop. Hence, it must be designed and developed correctly and efficiently.

One of the most popular object-oriented analysis and design methodologies is the Object Modeling Technique (OMT) developed slightly differently and independently by Jim Rumbaugh and his associates at General Electric R&D Center, and by Grady Booch at Rational Inc., CA. The two joined forces at Rational and developed the methodology (or notation) that today goes by the name Unified Modeling Language (UML). Coverage of UML is outside the scope of this book.

We will next look at some of the major components of software engineering including the role of OMT.

Product Specifications: This is and should be one of very first steps. The document that contains the specifications must be as complete as possible since the entire software development effort is based on what is contained in this document. It is rarely possible to arrive at a document that remains unchanged over the course of the project. When changes are made to the specification document, the reasons for the change should also be recorded. This can be very useful when the software undergoes periodic maintenance. In the context of software for engineering applications, it is important to include the details of the engineering theory that the software is supposed to incorporate.

Sometimes, it is as important to include in the specifications, what the software will not perform as it is important to spell out unambiguously, what the software will do. Here are some suggested steps in prescribing the product specifications.

- (1) Create a diagram that will show the overall flow of information through the program. Identify the input to the system and the output created by the system.
- (2) Now zoom in on the components from Step (1) and for each component identify the input and output, data storage schemes, theoretical details, etc.
- (3) Specify what are the software requirements in terms of software and hardware. Decide what are the limitations and assumptions, how exception handling is to be detected and handled.

¹ Adapted from Software Engineering Institute (SEI), CMU, Pittsburgh, PA.

Scheduling: Most software projects are team driven. To lay the groundwork for meeting the project deadlines, an appropriate procedure is used to (a) identify the different programming tasks, (b) estimate the amount of time required, and (c) the number of personnel involved directly or indirectly with the software development task. A software process model is typically used to control the activities. The scheduling is based on such methods as Program Evaluation and Review Technique (PERT) and Critical Path Method (CPM). Details of these techniques can be found in a book of software engineering.

Object-Oriented Modeling (OOM) and Design (OOD): Before we build an object-oriented system, we have to identify the classes associated with the problem. In addition, we also have to define how the different classes interact with each other. One commonly used approach is to describe the *use-cases* associated with the problem. A *use case* is a collection of possible sequences of interactions between the system under discussion and its external actors, related to a particular goal [Alistair Cockburn, 2000]. In other words, we need to describe how the actors (people, machines etc.) interact with the product to be built. The Class-Responsibility-Collaborator (CRC) modeling translates the information contained in *use-cases* into a representation of classes and their collaborations with other classes [Pressman, 2000]. What does this mean? We will use an example to answer this question and detail the OOM process.

We will start the process by examining the problem (or project) statement. Objects are identified by listing all the nouns or noun clauses. If the object is required to implement a solution, then it is part of the solution space.

Case Study: Steel Beam Cross-Section Selection Program

Problem Statement: You are required to design and operate a “AISC Steel Beam Cross-Section Selection” program. The beam cross-section has a symmetric I-section. The user of the system enters (a) the largest moment, (b) the largest tensile force, and (c) the largest compressive force, the beam is subjected to. The system finds the lightest cross-section from the available cross-sections database that will meet the strength (stress) requirements. Note that the selection is based on satisfying the following requirements for maximum compressive and tensile stresses.

$$\sigma_{\max}^c = \frac{|N_c|}{A} + \frac{|M|}{S} \leq 20000 \text{ psi}$$

$$\sigma_{\max}^t = \frac{|N_t|}{A} + \frac{|M|}{S} \leq 20000 \text{ psi}$$

where N_c is the largest compressive force, N_t is the largest tensile force, M is the largest bending moment, A is the cross-sectional area of the cross-section, and $S = \frac{I}{y_{\max}}$ is the section modulus (I is the moment of inertia). A sample I-section is shown in Fig. 13.1.1.

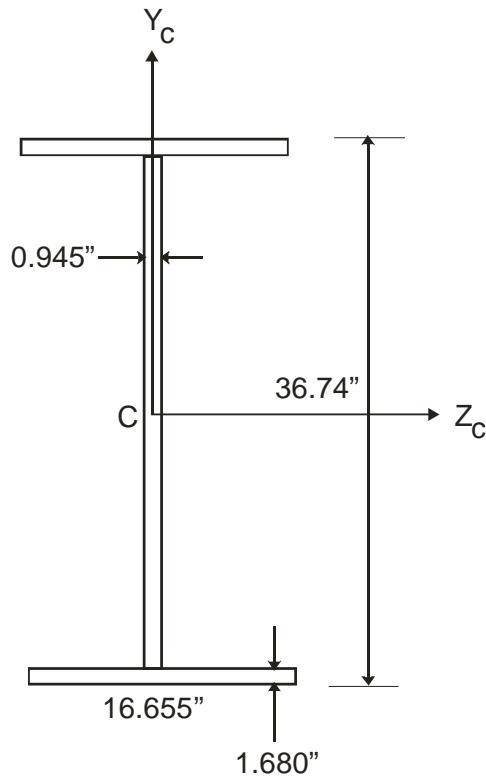


Fig. 13.1.1 W36x360 I-section

Solution: Analyzing the problem statement, we can identify the following nouns or noun clauses.

Beam	I-Section	User Input
Largest Moment	Tensile Force	Compressive Force
Lightest Cross-section	Cross-sections Database	

The next step in the process is to identify the different classes that model the system. A closer examination of these nouns will show the following. The I-Section is in fact the beam cross-section that we are after. All the different cross-sections are in a Cross-sections Database. The User Input contains the Largest Moment, Tensile Force and Compressive Force. Once we recognize how to use the user input, we can generate the criterion to locate the Lightest Cross-section.

We could use a class `CISection` to store the properties of a typical I-section such as the cross-sectional area, moment of inertia, etc. We could store the properties of all the I-sections using a class `CXSDatabase`. The class would allow access to all the individual I-sections, provide the mechanism to add new sections, or to delete sections that are no longer manufactured. To allow the selection of the lightest cross-section from the I-section database, we could define and use a new class `CXSSelector`.

Let's review our plan of action to convince ourselves that these classes are the major classes to achieve our objectives. The program would use the CXSDatabase class to load all the information about existing I-sections. It would then ask the user for the input and based on the input compute the 'smallest' properties of the I-section necessary to meet the strength requirements. Then the CXSSelector class would be used to find the lightest I-section (we will assume that the I-section database is arranged in order of increasing weights). Finally, the answer (the properties of the I-section) will be communicated to the user using the CISection class.

The next step is to define the responsibilities of each of these classes and the helper classes. The responsibilities of each class, as we have looked at before, is to define what each class is capable of doing – identify the attributes and behavior through member functions and variables. Sometimes, achieving these objectives requires the help of other classes – the helper classes. The CRC cards describing the classes are shown in Fig. 13.1.1.

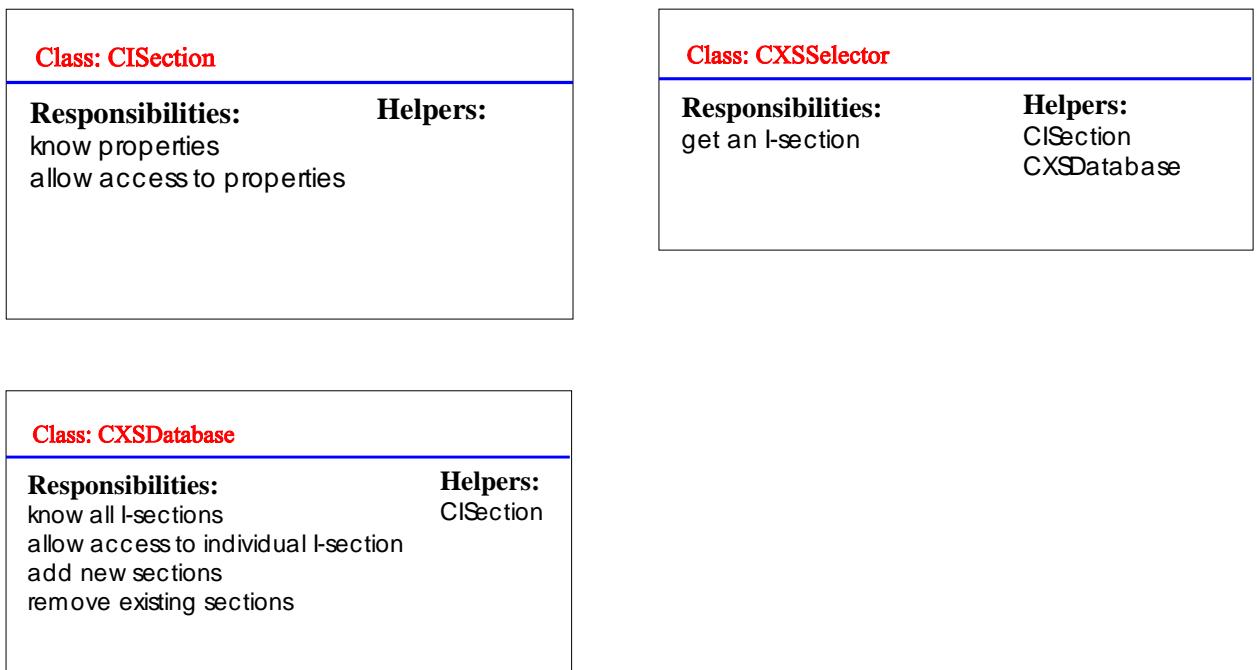


Fig. 13.1.1 Program classes described using CRC cards (Iteration 1)

Using the CRC cards as a guide, we can start the next step of defining the class members and variables. The three classes are described next.

```
class CISection
{
    public:
        CISection ();
        ~CISection ();
        // accessor functions
        ...
        // modifier functions
}
```

```

    ...
private:
    string m_szId;      // identification tag
    float m_fArea;      // x/s area
    float m_fSyy;        // section modulus y-axis
    float m_fSzz;        // section modulus z-axis
    float m_fWeight;     // weight per unit length
};

class CXSDatabase
{
public:
    CXSDatabase ();
    ~CXSDatabase ();
    // accessor functions
    ...
    // modifier functions
    void AddXSection (...);
    void RemoveXSection (...);
private:
    vector<CISection> m_AvailableSections;      // list of sections
    int m_nSize;                                  // # of sections
};

```

Note that the `std::vector` class is used to store the list of available sections since we need to dynamically change the size of the vector (that contains a dynamically allocated pointer-based data type, the `std::string` variable `m_szID`) – a feature not available with the `CVector` class.

```

class CXSSelector
{
public:
    CXSSelector ();
    ~CXSSelector ();
    // accessor functions
    CISection GetXSection(...);      // select a section
    // modifier functions
    ...
private:
    ...
};

```

We seem to have defined all the major classes. But how we? How do we obtain the user input? How do we process the user input? One way is to define a new class `CWizard` that would solicit the required input from the user, process the input, and display the results. This new class is shown in Fig. 13.1.2.

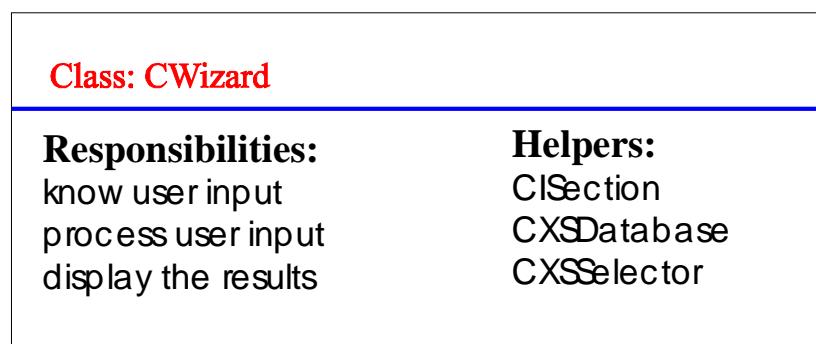


Fig. 13.1.2 The CWizard class description

```
class CWizard
{
public:
    CWizard ();
    ~Wizard ();
    // accessor functions
    GetUserInput ();
    ProcessUserInput ();
    DisplayResults ();
    // modifier functions
    ...
private:
};
```

Our initial design seems to be complete. Once all the classes are identified and defined, it is necessary to build a blueprint or a software architecture. The blueprint can then be used by a software developer to “put the pieces together” so as to build the software system.

Tip: Objects need identification. Sometimes the identification needs to be unique. For example, every person in the US has a unique identification that is the Social Security Number. The attributes that are associated with the identification constitute the *key*. In other words, the *key* is the identification tag. The data type associated with the *key* plays an important role in the manner in which the object is stored.

In the context of this exercise, we should recognize that there may be tens and hundreds of I-sections. How do we differentiate one I-section from the next? This is usually done through the section identifier, e.g. W36x300. In the context of the class definition, the variable `m_szID` is the key. We will see more about keys and classes later.

Object-Oriented Testing: As we can see even with the simple example discussed in this section, the chances of making errors – modeling, design and coding errors, are plentiful.

Once the initial blueprint is completed and the coding phase has started, testing should begin immediately. First, the classes can be tested individually. Then collaborating classes can be tested. In keeping with this trend, subsystems formed by several collaborating classes can be tested. If it is found

that the current state of the model and design is incomplete, incorrect, or extraneous, the modeling and design may have to be redone.

Example Program 13.1.1 Testing the CISection class

```

1  #include "isection.h"
2
3  int main ()
4  {
5      CISection First ("W36X300", 88.3f, 1110.0f, 156.0f, 300.0f);
6      First.Display ();
7
8      CISection Second ("W21X201", 59.2f, 461.0f, 86.1f, 201.0f);
9      Second.Display ();
10
11     CISection Third;
12     Third.Set ("W12X336", 98.8f, 483.0f, 177.0f, 336.0f);
13     Third.Display ();
14
15     CISection Fourth;
16     Fourth.SetProperties (Second);
17     Fourth.Display ();
18
19     return 0;
20 }
```

The design of the `CISection` class has undergone a few modifications since our initial design. There is a new constructor as follows. This is needed to instantiate an object when the I-section properties are known or obtained from user input.

```
CISection (const string, const float, const float, const float,
           const float);
```

This constructor is used in lines 5 and 8 to define two instances of the `CISection` class through the objects `First` and `Second`. The new `public` member functions are shown below.

```

// helper functions
void Display ();
// accessor functions
void GetProperties (string&, float&, float&, float&, float&);
// modifier functions
void SetProperties (const string, const float, const float,
                     const float, const float);
void SetProperties (const CISection&);
```

To test the original constructor, we use line 11 followed by the use of the member functions `SetProperties` and `Display`. The modifier function `SetProperties` is overloaded and tested in lines 12 and 16. The changes are not radical; they are made to make the class more versatile. As we have seen before, almost all classes have accessor, modifier and helper functions.

Example 13.1.2 Testing the CXSDatabase class

Our original class definition needs to undergo a few changes. Populating the database can be a tedious task since the database can potentially contain a large number of I-sections. Why not use an input file that has the values for the different I-sections and use the input file to populate the CXSDatabase object? To read the input from a file, we define a private variable `m_IFile`.

```
std::ifstream m_IFile; // (file) source of database
```

The constructor is modified to read the data from a predefined file, `XSdatabase.dat`.

```

1  CXSDatabase::CXSDatabase ()
2  {
3      // initialize
4      m_nSize = 0;
5      // open database file
6      m_IFile.open ("XSdatabase.dat", std::ios::in);
7      // load database
8      if (!m_IFile) {
9          cout << "Unable to open database file." << endl;
10         exit (0);
11     }
12     // read the data
13     string szID;
14     float fArea, fSyy, fSzz, fWeight;
15     CISection ISection;
16     for (;;) {
17         m_IFile >> szID >> fArea >> fSyy >> fSzz >> fWeight;
18         if (m_IFile.eof()) break; // end of file reached?
19         ISection.SetProperties (szID, fArea, fSyy, fSzz,
20                                fWeight);
21         Add (ISection);
22     }
23     m_IFile.close ();
24 }
25 }
```

The next change is a minor one. We need a `public` function to know how many cross-sections are available in the database (function `GetSize`) as well as access data associated with one of the cross-sections (function `GetOne`) through an integer that contains the vector index.

```

// helper functions
CISection GetOne (int) const;
// accessor functions
int GetSize () const;
```

We are now ready to test the class.

```

1 #include "xsdatabase.h"
2 #include "../Example13_1_1/isection.h"
3
4 int main ()
5 {
```

```

6      CXSDatabase DBISection; // load the 3 x/s
7      CISection New ("W12X336",98.8f,483.0f,177.0f,336.0f);
8
9      DBISection.Add (New);
10
11     CISection FoundOne = DBISection.GetOne(0);
12     FoundOne.Display ();
13
14     FoundOne = DBISection.GetOne(3);
15     FoundOne.Display ();
16
17     return 0;
18 }
```

Line 6 instantiates the object DBISection and the cross-sections database is read and stored in memory. A sample database file is shown below.

```

W21X201 59.2 461.0 86.1 201.0
W36X300 88.3 1110.0 156.0 300.0
W44X335 98.3 1410.0 150.0 335.0
```

The Add function is tested via lines 11, 14 and 15.

Finally, we are ready to test the entire program. We will test the CXSSelector class and the CWizard class via the final program.

Example Program 13.1.3 Testing the Wizard

Once again we will modify and enhance the original class definitions. Now that we know what is needed to select a particular cross-section from the database, we can add the parameters to the only public member function GetXSection in the CXSSelector class.

```

// accessor functions
int GetXSection(const CXSDatabase&,
                 CISection&, float, float, float);
```

To get the user input we need to define the variables that will store the user input as follows in the CWizard class.

```

private:
    CXSDatabase m_DBISection;
    CISection m_ISection;
    CXSSelector m_SelectLightest;
    float m_fBMMax, m_fTFMax, m_fCFMax;
    int m_nFound;
```

The CWizard holds all the data associated with the program via the variables – m_DBISection, m_ISection, m_SelectLightest. The variables m_fBMMax, m_fTFMax, m_fCFMax are the variables to store the user input values – maximum bending moment, maximum

tensile force, and maximum compressive force. The variable `m_found` is used to store the status of the search – whether a section has been found that satisfies all the strength requirements or not.

```

1 #include "wizard.h"
2
3 int main ()
4 {
5     CWizard theWizard;
6
7     while (theWizard.GetUserInput ())
8     {
9         theWizard.ProcessUserInput ();
10        theWizard.DisplayResults ();
11    }
12
13    return 0;
14 }
```

The main program is a very short one. The program is in a continuous loop until the user decides that there is no further input to the program. The `ProcessUserInput` function calls the `GetXSection` function in the `CXSSelector` class. It returns a positive value if a section is found that satisfies the strength requirements or a zero value if one if not found.

```

1 void CWizard::ProcessUserInput ()
2 {
3     m_nFound = m_SelectLightest.GetXSection (m_DBISection, m_ISection,
4                                         m_fBMMax, m_fTFMax, m_fCFMax);
5 }
6
7 void CWizard::DisplayResults ()
8 {
9     if (!m_nFound)
10         cout << endl << "No section satisfies the requirements.\n";
11     else {
12         cout << endl << "Lightest section:" ;
13         m_ISection.Display ();
14     }
15 }
```

A sample program execution is shown in Fig. 13.1.3.

Observation: Software development is an evolutionary process. The specifications, classes, and program architecture are likely to change over time. It is important to integrate the testing and prototyping process early on in the software development cycle.

We will see the development of more complex programs in later chapters.

```

MS-DOS "D:\winword\BookVer1\programs\Example10_6_3\Debug\Example10_6_3.exe"
I-Section Selection
(c)2000, S. D. Rajan

Any more requests to process (Y or N)? Y
Input positive values only.
Bending moment (lb-in): 200000
Tensile Force (lb): 10000
Compressive Force (lb): 0

Lightest section:
ID : W21X201
Area : 59.2 in^2
Section Modulus yy: 461 in^3
Section Modulus zz: 86.1 in^3
Weight : 201 lb/ft

Any more requests to process (Y or N)? N
Press any key to continue

```

Fig. 13.1.3 Sample session from Example Program 13.1.3 program

13.2 Inheritance

Object-oriented methodology provides building blocks. One such building block process is known as inheritance in which a new class called the derived (or child) class is created by building that class on top of an existing class called the base (or parent) class. The derived class then has all the member variables and ordinary member functions from the base class and can, in addition, define more variables and functions. Let's understand this concept.

Fig. 13.2.1 shows five different cross-sectional shapes that can be used as structural members. All the five cross-sectional shapes have the same set of (derived) attributes – cross-sectional area, moments of inertia, section modulus, and other similar properties. However, they are described differently. For example, a circular cross-section is defined in terms of a single attribute – radius, whereas a rectangular hollow section is described in terms of four attributes (Fig. 13.2.1(c)).

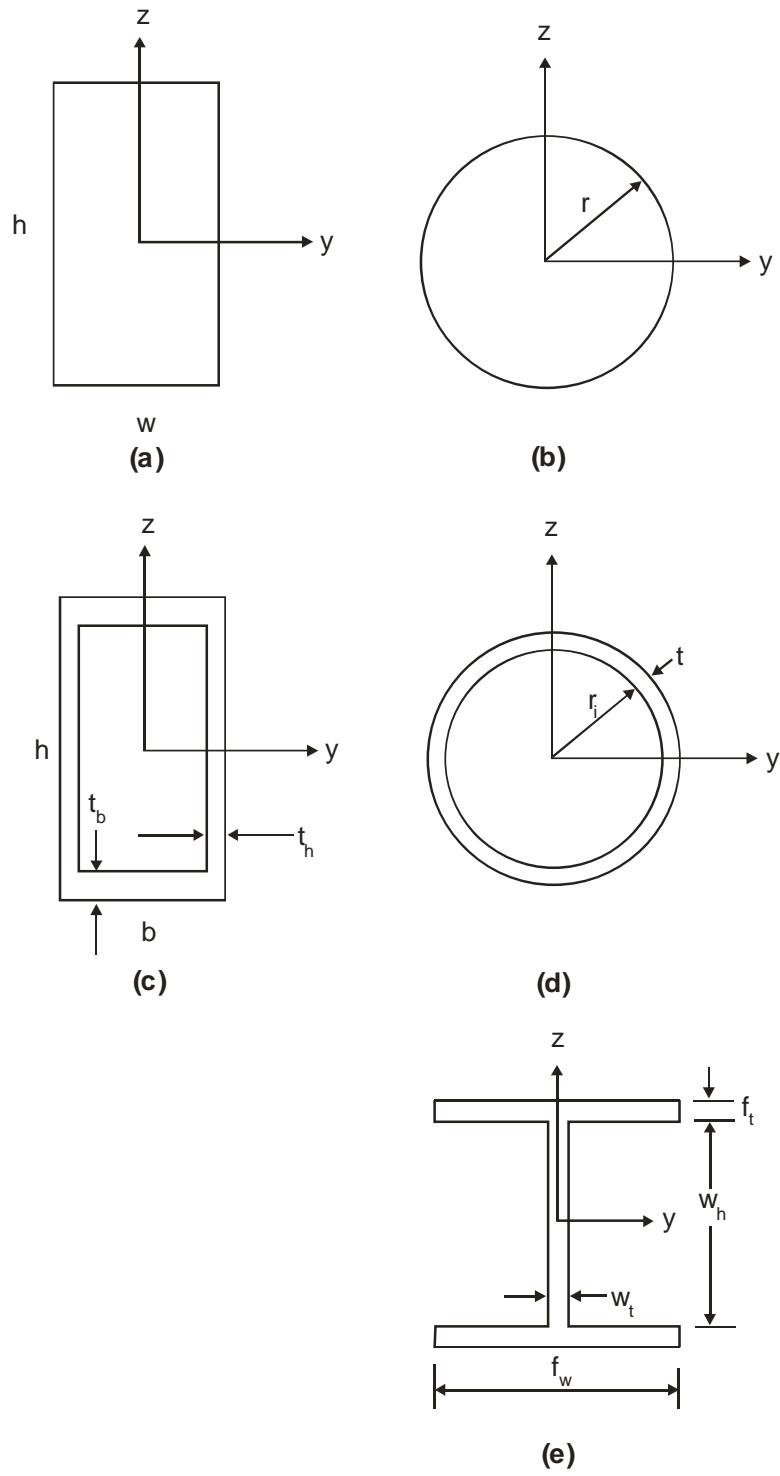


Fig. 13.2.1 Suite of cross-sections (a) Rectangular solid (b) Circular solid (c) Rectangular hollow (d) Circular hollow (e) Symmetric I-section

Inheritance provides the means of enhancing the capabilities of existing classes. If a new class is to be defined and an existing class is available, the new class can inherit the properties from the existing class. For example, we can define a *base class*, CXSType, for all cross-sectional shapes. This class would store and make available such things as the cross-section identification and the sectional properties. We could then define *derived classes* for different shapes such as CISection for I-sections and CCircSolid for circular solid cross-sections. If later, we need to add a new shape – rectangular solid, we could simply define a new derived class inheriting the properties of the base class, CXSType. The inheritance diagram for the cross-sections is shown in Fig. 13.2.2.

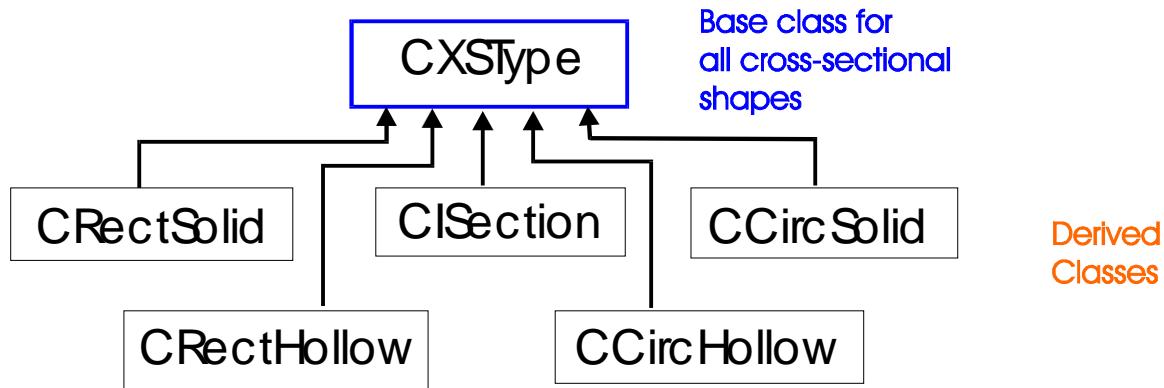


Fig. 13.2.2 Inheritance diagram

In this example, the base class contains what is generic to all cross-sectional shapes. Base classes are sometimes called *abstract* classes since base class objects are usually not constructed. CXSType is an abstract class since in a program we will not define an object directly tied to this class. However, we will define and use the constructor to store pertinent data associated with the class – the cross-section identification, the cross-sectional properties such as area and so on, and the cross-sectional dimensions (for example, height and width for a rectangular section). The base class also provides the generic accessor and modifier functions for both the cross-sectional properties and dimensions. What the base class **cannot** provide are the functions for computing the cross-sectional properties using the cross-sectional dimensions, and displaying the cross-sectional dimensions. This is because these are dependent on the cross-sectional shape that the base class does not know. Each cross-section has the following attributes and properties.

- (1) An identification tag, e.g. W36x300.
- (2) The cross-section area.
- (3) The section modulus about the (local) y and z axes.
- (4) The number of dimensions associated with the cross-section and the dimension values stored in a vector.

This information can be stored for every cross-section in the base class. However, the base class has no way of knowing how to compute the cross-sectional properties that must be computed separately by

each cross-section type. For the sake of simplicity, we will ignore the issue of units and assume that the values are stored in consistent units (inches). The base class, CXSType, is shown below.

Listing of the CXSType class (xstype.h)

```
class CXSType
{
public:
    CXSType ();
    CXSType (int);
    ~CXSType ();
    // helper functions
    void DisplayProperties ();
    void DisplayDimensions ();
    // accessor functions
    void GetProperties (std::string&, float&, float&,
                        float&);
    void GetDimensions (std::string&, CVector<float>&);

private:
    void Initialize (); // initializes all member variables

protected:
    std::string m_szID; // identification tag
    float m_fArea; // x/s area
    float m_fSyy; // section modulus y-axis
    float m_fSz; // section modulus z-axis
    int m_numDimensions; // number of dimensions
    CVector<float> m_fVDimensions; // dimensions
};
```

Note the use of **protected** qualifier. Earlier we saw two other qualifiers – **public** and **private**. This **protected** qualifier enables only the derived classes to access these member variables and functions of the base class. Using **private** would preclude that possibility. Using this base class definition, we can define the derived class. The derived class, CISection, is shown below.

Listing of the CISection class (isection.h)

```
#include "xstype.h"

const int numISDimensions = 4;

#include <string>

class CISection: public CXSType
{
public:
    CISection (const std::string&, const CVector<float>& fV);
    ~CISection ();
    // helper functions
    void DisplayDimensions ();

private:
    void ComputeProperties ();
};
```

With this definition, the CISection class inherits all the ordinary member functions in the CXSType class such as `DisplayProperties()`, `GetDimensions()` etc. and the member variables `m_szID`, `m_fArea` etc. The specialized member functions of a class such as the constructor, the destructor, the copy constructor and the assignment operator `=` are not inherited. A derived class can also have a member function with the same name and parameter types as the base class. This is called **redefining**² the inherited member function. In this example, the `DisplayDimensions()` function is redefined. As we will see later, the context in which the use (in the client code) takes place determines which function (base or derived) gets called.

Note how the derived class is defined. There is a colon after the derived class name followed by the keyword `public` and the base class name.

As mentioned earlier, the constructor in the base class is not inherited. When the derived class is instantiated, one should instantiate the base class explicitly. If the base class is not explicitly invoked, then the default base class constructor is called. A code cannot be compiled if the default base class constructor is not defined. Here is the code from the derived class, `CISection`, that shows the instantiation of derived class using an overloaded version of the base class.

```
CISection::CISection (const string& szID,
                      const CVector<float>& fV) : CXSType (numISDimensions)
{
    m_szID = szID;
    m_fVDimensions.SetSize (numISDimensions);
    for (int i=1; i <= numISDimensions; i++)
        m_fVDimensions(i) = fV(i);
    ComputeProperties ();
}
```

The statement

```
CISection::CISection (const string& szID,
                      const CVector<float>& fV) : CXSType (numISDimensions)
```

shows how the base class is instantiated using the overloaded `CISection` constructor. The base class overloaded constructor is called first followed by the derived class constructor.

```
CXSType::CXSType (int numDimensions)
{
    m_numDimensions = numDimensions;
    m_szID = "Undefined";
    m_fArea = 0.0f;
    m_fSyy = 0.0f;
    m_fSz = 0.0f;
}
```

² There is a difference between redefining and overloading. If there are two or more functions with the same name in a derived class, then the functions are overloaded. Similarly, if there are two or more functions with the same name in a base class, then the functions are overloaded. If there are functions in the base class and derived class with the same name but different parameters, then the base class function is redefined.

The design of this class is such that the cross-sectional properties are computed as soon as the cross-section is defined in terms of its ID and its dimensions. This is the reason why the default constructors are not provided in the derived classes since there is no way to call the derived class's function to compute the cross-sectional properties. In the next section, we will see how this potential problem can be avoided.

One should remember that if class B is derived from class A, then instantiating a class B object would involve class A constructor being invoked first followed by class B constructor. In the same vein, if object B goes out of scope, the destructor for class B is called first followed by the destructor for class A.

In the following example we show how the cross-sections inheritance concept is used in a client code.

Example Program 13.2.1 Using inheritance with cross-sectional shapes

xstype.cpp

```

1  #include <iostream>
2  #include <cassert>
3  #include "xstype.h"
4
5  CXSType::CXSType ()
6  {
7      Initialize ();
8  }
9
10 CXSType::CXSType (int numDimensions)
11 {
12     Initialize ();
13     m_numDimensions = numDimensions;
14     m_fVDimensions.SetSize (m_numDimensions);
15 }
16
17 void CXSType::Initialize ()
18 {
19     m_numDimensions = 0;
20     m_szID = "Undefined";
21     m_fArea = 0.0f;
22     m_fSyy = 0.0f;
23     m_fSzz = 0.0f;
24 }
25
26 CXSType::~CXSType ()
27 {
28 }
29
30 void CXSType::DisplayProperties ()
31 {
32     std::cout << '\n'
33         << "ID" : " << m_szID << '\n'
34         << "Area" : " << m_fArea << " in^2\n"
35         << "Section Modulus yy: " << m_fSyy << " in^3\n"

```

```

36         << "Section Modulus zz: " << m_fSzz     << " in^3\n";
37     }
38
39 void CXSType::DisplayDimensions ()
40 {
41     std::cout << "Invalid call to base class DisplayDimensions(). "
42                     << "Do not know what derived class dimensions mean.\n";
43 }
44
45 void CXSType::GetProperties (std::string& szID, float& fArea,
46                             float& fSyy, float& fSzz)
47 {
48     szID = m_szID;
49     fArea = m_fArea;
50     fSyy = m_fSyy;
51     fSzz = m_fSzz;
52 }
53
54 void CXSType::GetDimensions (std::string& szID, CVector<float>& fV)
55 {
56     assert (fV.GetSize() >= m_numDimensions);
57     szID = m_szID;
58     for (int i=1; i <= m_numDimensions; i++)
59         fV(i) = m_fVDimensions(i);
60 }
61

```

Note how the base class provides several functionalities that are common for all possible cross-sectional shapes. There are two constructors that initialize the values of the cross-sectional dimensions and the ID. The overloaded constructor also allocates memory to store the values of the cross-sectional dimensions. The `DisplayProperties()` member function displays the values of the cross-sectional properties that are common to all cross-sections. However, the `DisplayDimensions()` function displays an error message since the base class does not know the meaning of these dimensions. It would be inappropriate to call this function. The derived classes' `DisplayDimensions()` function should be called.

Next we show a sample client code.

main.cpp

```

1 #include "../library/vectortemplate.h"
2 #include "isection.h"
3 #include "rectsolid.h"
4
5 int main ()
6 {
7     CVector<float> fV(4);
8
9     // define the I-Section dimensions
10    fV(1) = 19.77f;    // web height
11    fV(2) = 0.91f;    // web thickness
12    fV(3) = 12.575f;  // flange width
13    fV(4) = 1.63f;    // flange thickness
14

```

```

15      CISection ISection1 ("W21x201", fV);
16      ISection1.DisplayDimensions ();
17      ISection1.DisplayProperties ();
18
19      // define the Rect Solid dimensions
20      fV(1) = 20.0f;    // height
21      fV(2) = 10.0f;   // width
22
23      CRectSolid RectSolid1 ("R20x10", fV);
24      RectSolid1.DisplayDimensions ();
25      RectSolid1.DisplayProperties ();
26
27      return 0;
28  }

```

The program illustrates the use of public inheritance with the base class `CXSType` and the derived classes, `CISection` and `CRectSolid`. The base class is purely abstract – there is no object in the program that is associated with the base class! When the object `ISection1` is declared, it inherits all the data and the attributes of the `CXSType` class in addition to the additional data and attributes, if any, of the `CISection` class. The `DisplayProperties()` functionality is provided by the base class while the `DisplayDimensions()` is provided by each derived class.

Access to Base Class's Function

In lines 16 and 24, the derived class's version of `DisplayDimensions` is called. What if we wish to call the base class version for some reason. We can if we use the scope `::` operator. For example, if we replace line 16 with

```
ISection1.CXSType::DisplayDimensions();
```

would call the base class's (`CXSType`) version of `DisplayDimensions` function.

Other Type of Inheritances

Protected and Private: In the previous example, we saw the derived classes `CISection` and `CRectSolid` publicly derived from the `CXSType` base class.

```
class CISection: public CXSType
{
...
};
```

If on the other hand, we have the following declaration

```
class CISection: protected CXSType
{
...
};
```

then the public members in the base class (`CXSType`) are protected in the derived class. If the

inheritance is private, then all the members of the base class are inaccessible in the derived class. The `protected` and `private` inheritances are seldom used in practice.

Multiple: A class may be derived from more than one base class. We saw an example of this in Chapter 12 when we looked at C++ file stream classes. For example, the `iostream` class is inherited from two classes – `istream` and `ostream`. Here are the class definitions.

```
class istream // handles input
{
...
};

class ostream // handles output
{
...
};

// iostream handles both input and output
class iostream: public istream, public ostream
{
...
};
```

The general syntax is as follows.

```
class classname: i_mode base_classname1, i_mode base_classname2, ...
```

In the above syntax `i_mode` is the inheritance mode – `public`, `private` or `protected`. As usual, if `i_mode` is not specified, the inheritance is assumed to be `private`. There is no limit to the number of base classes from which a class can be inherited. The same inheritance rules as we saw for a single inheritance apply to multiple inheritances. Multiple inheritances can lead to confusion over member function and variable names from different base classes. Almost always, multiple inheritances can be avoided through other means.

So how has inheritance helped? It has helped in several ways. First, it has forced us to think differently. We do not think of an I-section as being completely different than a rectangular solid section. We recognize that there are features that are common to all different sections and there are features that are unique to each section. Second, it helps us build a hierarchy starting with the definition of the base class that provides the gateway to all the features that are common to all the different cross-sectional shapes. This avoids unnecessary code duplication and maintenance. Third, it provides a mechanism to define, store and manipulate information that is unique to each cross-section.

Finally, let us examine what is necessary to define a new derived class from `CXSType`.

- The number of distinct cross-sectional dimensions.
- The formulae to use these cross-sectional dimensions to compute the cross-sectional properties.

- Constructor to allocate memory to store the dimension values.
- A function to display these cross-sectional dimensions.

What happens if the derived class does not provide some of these member functions? Can we streamline the inheritance mechanism to make the class more useful and avoid some of the pitfalls? We will see how this is done in the next section using the concept of *polymorphism*.

13.3 Polymorphism

In the previous section, we saw an example of inheritance with cross-sectional shapes. We asked the question as to what would happen if (a) the client code does not know ahead of time what specific cross-sectionals shape are going to be used during a program run, (b) new cross-sectional shapes are to be added, and (c) the process must be robust such that undefined situations are detected? The answer lies in the idea associated with polymorphism.

Polymorphism is the ability of different objects to respond in their own way to the same message by means of virtual functions or late binding or dynamic binding. In the context of the cross-sectional shape example, we want the program to automatically call the function that would compute the cross-sectional properties for any cross-sectional shape. In other words, we want the appropriate `ComputeProperties()` to be called when that function is invoked. We can achieve this objective by declaring the `ComputeProperties()` function as a virtual function. When the C++ compiler encounters a virtual function, it generates instructions to call the appropriate version of the function based on the instance of the object that is encountered during program execution. For example, if a `CRectSolid` object is created during program execution and a call is made to `ComputeProperties`, then `CRectSolid`'s version of `ComputeProperties` is called.

Example Program 13.3.1 Using inheritance with virtual member functions

As we will see, the changes that we need to make to the program from the preceding section are small. We first start making changes to the base class.

Listing of the CXSType class (xstype.h)

```
class CXSType
{
public:
    CXSType ();
    CXSType (int);
    ~CXSType ();
    // helper function
    void DisplayProperties ();
    // accessor functions
    void GetProperties (std::string&, float&, float&, float&);
    void GetDimensions (std::string&, CVector<float>&);
    virtual void DisplayDimensions ();
    virtual void ComputeProperties ();
private:
    void Initialize ();
```

```

protected:
    std::string m_szID;           // identification tag
    float m_fArea;               // x/s area
    float m_fSyy;                // section modulus y-axis
    float m_fSzz;                // section modulus z-axis
    int m_numDimensions;         // number of dimensions
    CVector<float> m_fVDimensions; // the dimensions
};

```

There are two changes. We declare the `DisplayDimensions` and the `ComputeProperties` functions as virtual functions. The `virtual` keyword precedes the function declaration. If a function is declared to be virtual and the function is redefined in the derived class, then for any object associated with the derived class the derived class version of the virtual function is used not the base class version. Both these functions are declared as public functions so that they can be called directly by the client code. Since in this example, the use of the base class functions is not desirable, we have the following error messages embedded in those two functions.

```

void CXSType::DisplayDimensions ()
{
    std::cout << "Invalid call to base class DisplayDimensions(). "
              << "Do not know what derived class dimensions mean.\n";
}

void CXSType::ComputeProperties ()
{
    std::cout << "CXSType::ComputeProperties(): This function "
              << "should not be called.\n";
}

```

We will illustrate the changes that we need to make in the derived class by looking at the `CISection` class definition.

Listing of the `CISection` class (`isection.h`)

```

#include "xstype.h"

const int numISDimensions = 4;

#include <string>

class CISection: public CXSType
{
public:
    CISection (const std::string&, const CVector<float>& fV);
    ~CISection ();
    // helper functions
    virtual void DisplayDimensions ();
    virtual void ComputeProperties ();
private:
};

```

While not required by C++ standards, we have declared the two functions as `virtual` functions just to remind ourselves that these are virtual functions. C++ requires that the `virtual` keyword be used only in the base class. Note that both these functions are declared as public functions so that they can

be called directly by the client code. The `virtual` keyword should not be used in when defining the body of the member function.

```
void CISection::DisplayDimensions ()
{
    std::cout << '\n'
        << "      I-Section Dimensions" << '\n'
        << "          Section ID : " << m_szID << '\n'
        << "          Web height : " << m_fVDimensions(1) << " in\n"
        << "          Web thickness : " << m_fVDimensions(2) << " in\n"
        << "          Flange width : " << m_fVDimensions(3) << " in\n"
        << "Flange thickness : " << m_fVDimensions(4) << " in\n";
}

void CISection::ComputeProperties ()
{
    // web height
    float wH = m_fVDimensions(1);
    // web thickness
    float wT = m_fVDimensions(2);
    // flange width
    float fW = m_fVDimensions(3);
    // flange thickness
    float fT = m_fVDimensions(4);

    // cross-sectional area
    m_fArea = wH*wT + 2.0f*fW*fT;
    // section modulus y-axis
    float fOI, fII;
    fOI = (pow(wH+2.0f*fT, 3.0f) * fW)/12.0f;
    fII = 2.0f*(pow(wH, 3.0f) * (0.5f*(fW-wT)))/12.0f;
    m_fSyy = (fOI - fII)/(0.5f*wH+fT);
    // section modulus z-axis
    float fIzz = 2.0f*(pow(fW, 3.0f)*fT)/12.0f +
                  pow(wT, 3)*wH/12.0f;
    m_fSzZ = fIzz/(0.5f*fW);
}
```

Finally, we do not have to make any changes to the client code. In fact, this version of the program is more robust!

Pure Virtual Functions

In the previous example, the base class, `CXSType` defined two virtual member functions - `DisplayDimensions` and `ComputeProperties`. However, these functions could function meaningfully since the base class knows neither the details about the cross-sectional dimensions nor how the cross-sectional properties are computed from the cross-sectional dimensions. These functions merely print error messages. These base class functions are called only if the derived class fails to provide its own version of these functions. C++ provides a mechanism to force the derived class to provide its copy of these functions. If the base class declares these functions to be `pure virtual` functions, the derived class then must define its version of these functions. To define a member function as a pure virtual function one must define the function as follows.

```
virtual return_type functionname (...) = 0;
```

A base class containing one or more pure virtual functions is called an abstract class since an object cannot be directly associated with an incomplete class. Similarly, a derived class is also abstract if it has one or more pure virtual functions. We will now illustrate the definition and use of pure virtual class using the same cross-sections example.

Example Program 13.3.2 Using inheritance with pure virtual member functions

We will now improve the quality of the previous example by defining the virtual functions as pure virtual functions.

```
Listing of the CXSType class (xstype.h)
class CXSType
{
public:
    virtual void DisplayDimensions () = 0;
    virtual void ComputeProperties () = 0;
...
};
```

We will then delete the bodies of the two member functions (`DisplayDimensions` and `ComputeProperties`) from the base class (`xstype.cpp`). No changes need to take place in the code associated with the derived classes. We finally have a program that is just about right!

Pointers and Virtual Functions

C++ provides more powerful features with inheritance and virtual functions. Consider, for example, the cross-sections problem. Let us assume that the I-sections are made of steel and the Rectangular Solid sections are made of wood. Let us assume that an additional member variable is necessary to define a rectangular solid section – moisture content (% units). This is the maximum permissible moisture content in the wood for use as a structural member.

```
class CRectSolid: public CXSType
{
public:
    CRectSolid (const std::string&, const CVector<float>& fV,
                const float fMC);
    CRectSolid (const CRectSolid&);
    ~CRectSolid ();
    // helper functions
    virtual void DisplayDimensions ();
    virtual void ComputeProperties ();
    float GetMC ();
    void SetMC (const float);

private:
    float m_fMC; // moisture content
};
```

We have seen that every I-section or Rectangular Solid section is a cross-section. If we have the following statements in a program, the program compiles fine.

```
CVector<float> fVDim(2); fVDim(1) = 4.0f; fVDim(2) = 2.0f;
CXSType      xsection;
CRectSolid  rsolid ("R2x4", fVDim, 10.5f);
xsection = rsolid;
```

It is perfectly valid to assign a derived class object to a base class object³. However, there is *slicing* problem. The information that is available only in the derived class cannot be stored and made available in the base class. In other words, the moisture content value is not available (sliced off the derived class data). A statement such as

```
std::cout << "Moisture content is " << xsection.GetMC() << '\n';
```

will not compile since the `GetMC()` function is not visible to the base class object.

C++ provides a solution to this problem – manipulating derived class information via a base class object. For example, if we change the above code as follows then the statements compile and execute correctly.

```
CVector<float> fVDim(2); fVDim(1) = 4.0f; fVDim(2) = 2.0f;
CXSType      *pxsection;
CRectSolid  *prsolid = new CRectSolid("R2x4", fVDim, 10.5f);
pxsection = prsolid;

std::cout << "Moisture content is " << pxsection->GetMC() << '\n';
...
delete prsolid;
```

Finally, a point about destructors. When inheritance is used, it is a good idea to define virtual destructors. In the above code, when the statement

```
delete prsolid;
```

is executed, the destructor of class `CRectSolid` is called. If on the other hand the code had been written as follows

```
CVector<float> fVDim(2); fVDim(1) = 4.0f; fVDim(2) = 2.0f;
CXSType      *pxsection = new CRectSolid("R2x4", fVDim, 10.5f);

std::cout << "Moisture content is " << pxsection->GetMC() << '\n';
...
delete pxsection;
```

³ It should also be noted that a derived class object cannot be assigned to a base class object. For example, the following statement will not compile.

```
rsolid = xsection;
```

then the last statement would call the destructor of class CXSType. Since the destructor of class CRectSolid is not called, this process can be dangerous especially if resource deallocation takes place in the CRectSolid class. All the derived class destructors can be automatically called if the destructor for the base class is declared as a **virtual** destructor. In other words, if a base class destructor is tagged virtual, then the derived class destructor is called first followed by the base class destructor. It should also be noted that if the destructor for the base class is tagged virtual, then automatically destructors for the derived classes are tagged virtual. This scheme provides a very nice approach to information management as we will later see in the example in this section.

Example Program 13.3.3 Using inheritance with pure virtual member functions and virtual destructors

Develop a program that demonstrates how to dynamically generate, store and manipulate information dealing with cross-sections. The type of the cross-section and its associated data are known only at runtime. The design should be such that adding new types of cross-sections should have a well-defined easy and robust path.

The solution is built on all the discussions we have had in this preceding sections. We explain the solution by showing the developed source code.

xstype.h

```

1  #ifndef __CXSTYPE_H__
2  #define __CXSTYPE_H__
3
4  #include <string>
5  #include "../library/vectortemplate.h"
6
7  class CXSType
8  {
9      public:
10         CXSType ();
11         CXSType (int);
12         virtual ~CXSType ();
13         // helper function
14         void DisplayProperties ();
15         // accessor functions
16         void GetProperties (std::string&, float&, float&, float&);
17         void GetDimensions (std::string&, CVector<float>&);
18         virtual void DisplayDimensions () = 0;
19         virtual void ComputeProperties () = 0;
20     private:
21         void Initialize ();
22     protected:
23         std::string m_szID;           // identification tag
24         float m_fArea;              // x/s area
25         float m_fSyy;               // section modulus y-axis
26         float m_fSzz;               // section modulus z-axis
27         int m_numDimensions;        // number of dimensions
28         CVector<float> m_fVDimensions; // the dimensions
29     };

```

```

30
31 #endif

```

Compared to the earlier uses of the CXType class, there are no changes to the definition of the base class except to declare the destructor as a virtual destructor. This has major implications as we discussed earlier.

xstype.cpp

```

1  #include <iostream>
2  #include <cassert>
3  #include "xstype.h"
4
5  CXSType::CXSType ()
6  {
7      Initialize ();
8  }
9
10 CXSType::CXSType (int numDimensions)
11 {
12     Initialize ();
13     m_numDimensions = numDimensions;
14     m_fVDimensions.SetSize (m_numDimensions);
15 }
16
17 void CXSType::Initialize ()
18 {
19     m_numDimensions = 0;
20     m_szID = "Undefined";
21     m_fArea = 0.0f;
22     m_fSy = 0.0f;
23     m_fSz = 0.0f;
24 }
25
26 CXSType::~CXSType ()
27 {
28 }
29
30 void CXSType::DisplayProperties ()
31 {
32     ComputeProperties ();
33     std::cout << '\n'
34         << "ID" : " << m_szID << '\n'
35         << "Area" : " << m_fArea << " in^2\n"
36         << "Section Modulus yy: " << m_fSy << " in^3\n"
37         << "Section Modulus zz: " << m_fSz << " in^3\n";
38 }
39
40 void CXSType::GetProperties (std::string& szID, float& fArea,
41                             float& fSy, float& fSz)
42 {
43     ComputeProperties ();
44     szID = m_szID;
45     fArea = m_fArea;
46     fSy = m_fSy;

```

```

47     fSzz = m_fSzz;
48 }
49
50 void CXSType::GetDimensions (std::string& szID, CVector<float>& fv)
51 {
52     assert (fv.GetSize() >= m_numDimensions);
53     szID = m_szID;
54     for (int i=1; i <= m_numDimensions; i++)
55         fv(i) = m_fVDimensions(i);
56 }
```

Compared to the earlier uses of the CXType class, there are no changes to the definition of the base class.

rectsolid.h

```

1 #ifndef __RAJAN_RECTSOLID_H__
2 #define __RAJAN_RECTSOLID_H__
3
4 #include <string>
5 #include "xstype.h"
6 const int numRectDimensions = 2;
7
8 class CRectSolid: public CXSType
9 {
10     public:
11         CRectSolid (const std::string&, const CVector<float>& fv,
12                     const float fMC);
13         CRectSolid (const CRectSolid&);
14         ~CRectSolid ();
15         // helper functions
16         virtual void DisplayDimensions ();
17         virtual void ComputeProperties ();
18         float GetMC ();
19         void SetMC (const float);
20
21     private:
22         float m_fMC; // moisture content
23     };
24
25 #endif
```

Compared to the earlier uses of the CRectSolid class, there are a few changes. We have added a private member variable, m_fMC to store the maximum moisture content. To handle this extra member variable, changes are made to the overloaded constructor (lines 11-12) and two new member functions are declared (lines 18 and 19).

rectsolid.cpp

```

1 #include <cmath>
2 #include <iostream>
3 #include "rectsolid.h"
4
```

```

5   CRectSolid::CRectSolid (const std::string& szID,
6           const CVector<float>& fV,
7           const float fMC) : CXSType (numRectDimensions)
8   {
9       assert (fV.GetSize() >= numRectDimensions);
10      m_szID = szID;
11      m_fMC = fMC;
12      for (int i=1; i <= numRectDimensions; i++)
13          m_fVDimensions(i) = fV(i);
14      ComputeProperties ();
15  }
16
17 CRectSolid::~CRectSolid ()
18 {
19 }
20
21 void CRectSolid::DisplayDimensions ()
22 {
23     std::cout << '\n'
24         << "    Rectangular Solid Dimensions and Traits\n"
25         << "        Section ID : " << m_szID << '\n'
26         << "        Height : " << m_fVDimensions(1) << " in\n"
27         << "        Width : " << m_fVDimensions(2) << " in\n"
28         << "        Moisture Content : " << m_fMC << "%\n";
29 }
30
31 void CRectSolid::ComputeProperties ()
32 {
33     // height
34     float fH = m_fVDimensions(1);
35     // width
36     float fW = m_fVDimensions(2);
37
38     // cross-sectional area
39     m_fArea = fH * fW;
40     // section modulus y-axis
41     m_fSyy = (pow(fH, 3.0f)*fW/12.0f)/(0.5f*fH);
42     // section modulus z-axis
43     m_fSzz = (pow(fW, 3.0f)*fH/12.0f)/(0.5f*fW);
44 }

```

Compared to the earlier uses of the `CRectSolid` class, there are a few changes. These relate to the new member variable, `m_fMC` that is handled by the overloaded constructor and two new member functions.

main.cpp

```

1 #include <iostream>
2 #include "../library/vectortemplate.h"
3 #include "isection.h"
4 #include "rectsolid.h"
5
6 int main ()
7 {
8     const int NUMSECTIONS = 2;

```

```

9      CVector<CXSType*> pVXSections(NUMSECTIONS);
10     CVector<float> fV(4);
11
12     // define the I-Section dimensions
13     fV(1) = 19.77f;      // web height
14     fV(2) = 0.91f;       // web thickness
15     fV(3) = 12.575f;    // flange width
16     fV(4) = 1.63f;      // flange thickness
17
18     pVXSections(1) = new CISection ("W21x201", fV);
19     pVXSections(1)->DisplayDimensions ();
20     pVXSections(1)->DisplayProperties ();
21
22     // define the Rect Solid dimensions
23     fV(1) = 20.0f;      // height
24     fV(2) = 10.0f;      // width
25
26     pVXSections(2) = new CRectSolid ("R20x10", fV, 10.5f);
27     pVXSections(2)->DisplayDimensions ();
28     pVXSections(2)->DisplayProperties ();
29
30     for (int i=1; i <= NUMSECTIONS; i++)
31         delete pVXSections(i);
32
33     return 0;
34 }
```

This client code merits a close look. We use the `CVector` class as a container. The basic idea is to use pointers to store the address of each cross-section as they are defined during execution in a vector (line 9). As we saw earlier in this section, base class pointers can be used to manipulate derived class objects. In this sample program we define only two cross-sections. There is no reason why the size of the vector cannot be set dynamically as we have done before in earlier chapters using the `SetSize` member function. Line 18 shows how memory is dynamically allocated to store one object of type `CISection`. After that point, the program is similar to the previous version. Line 26 shows another dynamic memory allocation to store another cross-section object of type `CRectSolid`. Finally, note memory deallocation must take place (so as to avoid memory leak) and statements in lines 30-31 achieve this objective.

Using Run-Time Typing Information

In the previous example we saw how a vector of base class pointers can be used to hold the address of derived class objects created at run time. What can we do if at run-time we need to know the type of the object so that an appropriate action can be taken? For example, let's look at the scenario when we need to print the moisture content of a cross-section. C++ provides a casting operator known as `dynamic_cast` that can be used as follows to handle the scenario.

```
// display moisture content
for (int i=1; i <= NUMSECTIONS; i++)
{
    if (CRectSolid *pRS = dynamic_cast<CRectSolid *>(pVXSections(i)))
    {
```

```

        std::cout << "Section : " << i << pRS->GetMC() << '\n';
    }
}
```

The template argument is a type and the argument is a pointer or a reference. In the above example, the type is `CRectSolid *` (a pointer to `CRectSolid`) and the argument is the base class pointer (`CXSType *`). If the type of the parameter value matches the template argument type, the value is returned; if the value does not match, a `NULL` pointer is returned. In the above example, if `pVXSections(i)` does **not** hold the address of a rectangular solid section (`CRectSolid *`), then the returned value is `NULL` and the `if` statement evaluates to `false`. This type of type casting is known as *upcasting* since the conversion moves up the class hierarchy (from the base class to a derived class). It should be noted that if a reference is used (instead of a pointer), the casting failure results in a `bad_cast` exception rather than a `NULL` pointer.

C++ also provides a `typeid` operator that can be used to obtain the type information. For example, this operator takes as the argument either an expression or a class name and returns an object of type `type_info`. Here are a couple of examples to illustrate the usage.

```

#include <typeinfo>
...
CRectSolid* pR1 = new CRectSolid ("R20x10", fV, 10.5f);
std::cout << "Typeid information: " << typeid(pR1).name() << '\n';
std::cout << "Typeid information: " << typeid(*pR1).name() << '\n';
```

The program displays the following output.

```

Typeid information: class CRectSolid *
Typeid information: class CRectSolid
```

The `typeid` operator can also be used to test or compare the `typeinfo` value.

```

#include <typeinfo>
...
CRectSolid* pR1 = new CRectSolid ("R20x10", fV, 10.5f);
if (typeid(pR1) == typeid(CRectSolid *))
    std::cout << "Cross section is a rectangular solid.\n";
```

Note that

```

if (typeid(pR1) == typeid(CRectSolid *))
```

is not the same as

```

if (typeid(pR1) == typeid(CRectSolid))
```

One should be cautious in defining virtual member functions. If design requires the use of virtual functions then go ahead and use them. Otherwise do not. There is an overhead associated with the usage of virtual functions. This overhead can sometimes slow down the execution of a program substantially. C++ compilers create a virtual function table for any class that uses virtual functions. The

address of each of these virtual functions is stored in the table. This table management can be quite involved if multiple inheritances are involved and member functions are overridden.

Nested classes

A class can be defined inside another class as long as there is no ambiguity. For example, the following class definition is not valid since class CA refers to class CB that is private.

```
class CA
{
    float m_fx;      // by default, private
    class CB { };   // by default, private
    class CC        // by default, private
    {
        CB m_b;
    };
};
```

Similarly, the member variable `m_fx` cannot be used inside class CB or class CC since once again, `m_fx` is a private member variable. In the above example, class CA is the enclosing class and classes CB and CC are nested classes. Nested classes are independent classes. An object of the nested type does not have members defined by the enclosing class. Conversely, an object of the enclosing class does not have members defined by the nested class. Here is an example that is syntactically correct.

```
class A      // enclosing or outer class
{
    private:
        float m_fx;
        class B { };   // private nested or inner class
    public:
        class C        // public nested or inner class
        {
            void xyz (int n); // private by default
            ...
            public:
                void ABC (int n);
        };
};
```

A nested class can be either public or private. If it is public, then it can be used outside the enclosing class. Using the above example, one can declare a variable as

```
A::C aCobject;
```

Nested classes are typically used to avoid naming conflicts – to place the name of a class inside the scope of another class so that two classes can contain another class inside them with the same name and having different functionalities.

Example Program 13.3.4 Nested Classes

Develop a program to store basic material properties and allow for conversion of values from one set of units to another (SI and US Customary units). Assume that the SI-related units are kg, m, s, and the USC-related units are slug, ft, s.

We will define a base class `CMaterial` designed to have member functions and variables to store the material data. We will derive two classes `CMaterialSI` and `CMaterialUSC` to define objects to store material properties in SI and US Customary units respectively. In each of these classes we will define a nested class `CConvert` that will convert the material values from SI to USC units and back.

material.h

```

1  #include <iostream>
2  #include <string>
3
4  // abstract base class
5  class CMaterial
6  {
7      public:
8          CMaterial();
9          ~CMaterial();
10         enum PROPERTY {MASSDENSITY=0, YIELDSTRENGTH, MODULUSOFElasticITY,
11                         LASTONE};
12
13     // modifier functions
14     void SetName (const std::string& szName);
15     void SetMassDensity (double dMassDensity);
16     void SetYieldStrength (double dYieldStrength);
17     void SetModulusOfElasticity (double dModulusOfElasticity);
18
19     // accessor functions
20     void GetName (std::string& szName) const;
21     double GetMassDensity () const;
22     double GetYieldStrength () const;
23     double GetModulusOfElasticity () const;
24
25     // helper function
26     void Display () const;
27
28     private:
29         std::string m_szName;
30         double      m_dMassDensity;
31         double      m_dYieldStrength;
32         double      m_dModulusOfElasticity;
33     };
34
35 // SI version: kg, m, s
36 class CMaterialSI : public CMaterial
37 {
38     public:
39         class CConvert
40         {
41             public:
42                 CConvert ();
43                 virtual ~CConvert();
44                 double GetValue (const CMaterialSI&, enum PROPERTY);
45         };
46         CMaterialSI ();
47         ~CMaterialSI ();
48         CMaterialSI (const std::string& szName, double dMassDensity,
49                      double dYieldStrength, double dModulusOfElasticity);
50     private:
51         static double m_dVConvFactor[LASTONE];
52     };
53
54 // SI version: slug, ft, s
55 class CMaterialUSC : public CMaterial
56 {
57     public:
58         class CConvert
59         {
60             public:
61                 CConvert ();
62         };
63
64         class CConvert
65         {
66             public:
67                 CConvert ();
68         };
69
70         class CConvert
71         {
72             public:
73                 CConvert ();
74         };
75
76         class CConvert
77         {
78             public:
79                 CConvert ();
80         };
81
82         class CConvert
83         {
84             public:
85                 CConvert ();
86         };
87
88         class CConvert
89         {
90             public:
91                 CConvert ();
92         };
93
94         class CConvert
95         {
96             public:
97                 CConvert ();
98         };
99
100        class CConvert
101        {
102            public:
103                CConvert ();
104        };
105
106        class CConvert
107        {
108            public:
109                CConvert ();
110        };
111
112        class CConvert
113        {
114            public:
115                CConvert ();
116        };
117
118        class CConvert
119        {
120            public:
121                CConvert ();
122        };
123
124        class CConvert
125        {
126            public:
127                CConvert ();
128        };
129
130        class CConvert
131        {
132            public:
133                CConvert ();
134        };
135
136        class CConvert
137        {
138            public:
139                CConvert ();
140        };
141
142        class CConvert
143        {
144            public:
145                CConvert ();
146        };
147
148        class CConvert
149        {
150            public:
151                CConvert ();
152        };
153
154        class CConvert
155        {
156            public:
157                CConvert ();
158        };
159
160        class CConvert
161        {
162            public:
163                CConvert ();
164        };
165
166        class CConvert
167        {
168            public:
169                CConvert ();
170        };
171
172        class CConvert
173        {
174            public:
175                CConvert ();
176        };
177
178        class CConvert
179        {
180            public:
181                CConvert ();
182        };
183
184        class CConvert
185        {
186            public:
187                CConvert ();
188        };
189
190        class CConvert
191        {
192            public:
193                CConvert ();
194        };
195
196        class CConvert
197        {
198            public:
199                CConvert ();
200        };
201
202        class CConvert
203        {
204            public:
205                CConvert ();
206        };
207
208        class CConvert
209        {
210            public:
211                CConvert ();
212        };
213
214        class CConvert
215        {
216            public:
217                CConvert ();
218        };
219
220        class CConvert
221        {
222            public:
223                CConvert ();
224        };
225
226        class CConvert
227        {
228            public:
229                CConvert ();
230        };
231
232        class CConvert
233        {
234            public:
235                CConvert ();
236        };
237
238        class CConvert
239        {
240            public:
241                CConvert ();
242        };
243
244        class CConvert
245        {
246            public:
247                CConvert ();
248        };
249
250        class CConvert
251        {
252            public:
253                CConvert ();
254        };
255
256        class CConvert
257        {
258            public:
259                CConvert ();
260        };
261
262        class CConvert
263        {
264            public:
265                CConvert ();
266        };
267
268        class CConvert
269        {
270            public:
271                CConvert ();
272        };
273
274        class CConvert
275        {
276            public:
277                CConvert ();
278        };
279
280        class CConvert
281        {
282            public:
283                CConvert ();
284        };
285
286        class CConvert
287        {
288            public:
289                CConvert ();
290        };
291
292        class CConvert
293        {
294            public:
295                CConvert ();
296        };
297
298        class CConvert
299        {
299            public:
300                CConvert ();
301        };
302
303        class CConvert
304        {
305            public:
306                CConvert ();
307        };
308
309        class CConvert
310        {
311            public:
312                CConvert ();
313        };
314
315        class CConvert
316        {
317            public:
318                CConvert ();
319        };
320
321        class CConvert
322        {
323            public:
324                CConvert ();
325        };
326
327        class CConvert
328        {
329            public:
330                CConvert ();
331        };
332
333        class CConvert
334        {
335            public:
336                CConvert ();
337        };
338
339        class CConvert
340        {
341            public:
342                CConvert ();
343        };
344
345        class CConvert
346        {
347            public:
348                CConvert ();
349        };
350
351        class CConvert
352        {
353            public:
354                CConvert ();
355        };
356
357        class CConvert
358        {
359            public:
360                CConvert ();
361        };
362
363        class CConvert
364        {
365            public:
366                CConvert ();
367        };
368
369        class CConvert
370        {
371            public:
372                CConvert ();
373        };
374
375        class CConvert
376        {
377            public:
378                CConvert ();
379        };
380
381        class CConvert
382        {
383            public:
384                CConvert ();
385        };
386
387        class CConvert
388        {
389            public:
390                CConvert ();
391        };
392
393        class CConvert
394        {
395            public:
396                CConvert ();
397        };
398
399        class CConvert
400        {
401            public:
402                CConvert ();
403        };
404
405        class CConvert
406        {
407            public:
408                CConvert ();
409        };
410
411        class CConvert
412        {
413            public:
414                CConvert ();
415        };
416
417        class CConvert
418        {
419            public:
420                CConvert ();
421        };
422
423        class CConvert
424        {
425            public:
426                CConvert ();
427        };
428
429        class CConvert
430        {
431            public:
432                CConvert ();
433        };
434
435        class CConvert
436        {
437            public:
438                CConvert ();
439        };
440
441        class CConvert
442        {
443            public:
444                CConvert ();
445        };
446
447        class CConvert
448        {
449            public:
450                CConvert ();
451        };
452
453        class CConvert
454        {
455            public:
456                CConvert ();
457        };
458
459        class CConvert
460        {
461            public:
462                CConvert ();
463        };
464
465        class CConvert
466        {
467            public:
468                CConvert ();
469        };
470
471        class CConvert
472        {
473            public:
474                CConvert ();
475        };
476
477        class CConvert
478        {
479            public:
480                CConvert ();
481        };
482
483        class CConvert
484        {
485            public:
486                CConvert ();
487        };
488
489        class CConvert
490        {
491            public:
492                CConvert ();
493        };
494
495        class CConvert
496        {
497            public:
498                CConvert ();
499        };
499
500        class CConvert
501        {
502            public:
503                CConvert ();
504        };
505
506        class CConvert
507        {
508            public:
509                CConvert ();
510        };
511
512        class CConvert
513        {
514            public:
515                CConvert ();
516        };
517
518        class CConvert
519        {
520            public:
521                CConvert ();
522        };
523
524        class CConvert
525        {
526            public:
527                CConvert ();
528        };
529
530        class CConvert
531        {
532            public:
533                CConvert ();
534        };
535
536        class CConvert
537        {
538            public:
539                CConvert ();
540        };
541
542        class CConvert
543        {
544            public:
545                CConvert ();
546        };
547
548        class CConvert
549        {
550            public:
551                CConvert ();
552        };
553
554        class CConvert
555        {
556            public:
557                CConvert ();
558        };
559
560        class CConvert
561        {
562            public:
563                CConvert ();
564        };
565
566        class CConvert
567        {
568            public:
569                CConvert ();
570        };
571
572        class CConvert
573        {
574            public:
575                CConvert ();
576        };
577
578        class CConvert
579        {
580            public:
581                CConvert ();
582        };
583
584        class CConvert
585        {
586            public:
587                CConvert ();
588        };
589
590        class CConvert
591        {
592            public:
593                CConvert ();
594        };
595
596        class CConvert
597        {
598            public:
599                CConvert ();
600        };
601
602        class CConvert
603        {
604            public:
605                CConvert ();
606        };
607
608        class CConvert
609        {
610            public:
611                CConvert ();
612        };
613
614        class CConvert
615        {
616            public:
617                CConvert ();
618        };
619
620        class CConvert
621        {
622            public:
623                CConvert ();
624        };
625
626        class CConvert
627        {
628            public:
629                CConvert ();
630        };
631
632        class CConvert
633        {
634            public:
635                CConvert ();
636        };
637
638        class CConvert
639        {
640            public:
641                CConvert ();
642        };
643
644        class CConvert
645        {
646            public:
647                CConvert ();
648        };
649
650        class CConvert
651        {
652            public:
653                CConvert ();
654        };
655
656        class CConvert
657        {
658            public:
659                CConvert ();
660        };
661
662        class CConvert
663        {
664            public:
665                CConvert ();
666        };
667
668        class CConvert
669        {
669            public:
670                CConvert ();
671        };
672
673        class CConvert
674        {
675            public:
676                CConvert ();
677        };
678
679        class CConvert
680        {
681            public:
682                CConvert ();
683        };
684
685        class CConvert
686        {
687            public:
688                CConvert ();
689        };
690
691        class CConvert
692        {
693            public:
694                CConvert ();
695        };
696
697        class CConvert
698        {
699            public:
700                CConvert ();
701        };
702
703        class CConvert
704        {
705            public:
706                CConvert ();
707        };
708
709        class CConvert
710        {
711            public:
712                CConvert ();
713        };
714
715        class CConvert
716        {
717            public:
718                CConvert ();
719        };
720
721        class CConvert
722        {
723            public:
724                CConvert ();
725        };
726
727        class CConvert
728        {
729            public:
730                CConvert ();
731        };
732
733        class CConvert
734        {
735            public:
736                CConvert ();
737        };
738
739        class CConvert
740        {
741            public:
742                CConvert ();
743        };
744
745        class CConvert
746        {
747            public:
748                CConvert ();
749        };
750
751        class CConvert
752        {
753            public:
754                CConvert ();
755        };
756
757        class CConvert
758        {
759            public:
760                CConvert ();
761        };
762
763        class CConvert
764        {
765            public:
766                CConvert ();
767        };
768
769        class CConvert
770        {
771            public:
772                CConvert ();
773        };
774
775        class CConvert
776        {
777            public:
778                CConvert ();
779        };
780
781        class CConvert
782        {
783            public:
784                CConvert ();
785        };
786
787        class CConvert
788        {
789            public:
790                CConvert ();
791        };
792
793        class CConvert
794        {
795            public:
796                CConvert ();
797        };
798
799        class CConvert
800        {
801            public:
802                CConvert ();
803        };
804
805        class CConvert
806        {
807            public:
808                CConvert ();
809        };
810
811        class CConvert
812        {
813            public:
814                CConvert ();
815        };
816
817        class CConvert
818        {
819            public:
820                CConvert ();
821        };
822
823        class CConvert
824        {
825            public:
826                CConvert ();
827        };
828
829        class CConvert
830        {
831            public:
832                CConvert ();
833        };
834
835        class CConvert
836        {
837            public:
838                CConvert ();
839        };
840
841        class CConvert
842        {
843            public:
844                CConvert ();
845        };
846
847        class CConvert
848        {
849            public:
850                CConvert ();
851        };
852
853        class CConvert
854        {
855            public:
856                CConvert ();
857        };
858
859        class CConvert
860        {
861            public:
862                CConvert ();
863        };
864
865        class CConvert
866        {
867            public:
868                CConvert ();
869        };
870
871        class CConvert
872        {
873            public:
874                CConvert ();
875        };
876
877        class CConvert
878        {
879            public:
880                CConvert ();
881        };
882
883        class CConvert
884        {
885            public:
886                CConvert ();
887        };
888
889        class CConvert
890        {
891            public:
892                CConvert ();
893        };
894
895        class CConvert
896        {
897            public:
898                CConvert ();
899        };
899
900        class CConvert
901        {
902            public:
903                CConvert ();
904        };
905
906        class CConvert
907        {
908            public:
909                CConvert ();
910        };
911
912        class CConvert
913        {
914            public:
915                CConvert ();
916        };
917
918        class CConvert
919        {
920            public:
921                CConvert ();
922        };
923
924        class CConvert
925        {
926            public:
927                CConvert ();
928        };
929
930        class CConvert
931        {
932            public:
933                CConvert ();
934        };
935
936        class CConvert
937        {
938            public:
939                CConvert ();
940        };
941
942        class CConvert
943        {
944            public:
945                CConvert ();
946        };
947
948        class CConvert
949        {
950            public:
951                CConvert ();
952        };
953
954        class CConvert
955        {
956            public:
957                CConvert ();
958        };
959
960        class CConvert
961        {
962            public:
963                CConvert ();
964        };
965
966        class CConvert
967        {
968            public:
969                CConvert ();
970        };
971
972        class CConvert
973        {
974            public:
975                CConvert ();
976        };
977
978        class CConvert
979        {
980            public:
981                CConvert ();
982        };
983
984        class CConvert
985        {
986            public:
987                CConvert ();
988        };
989
990        class CConvert
991        {
992            public:
993                CConvert ();
994        };
995
996        class CConvert
997        {
998            public:
999                CConvert ();
1000           CConvert ();
1001
1002           CConvert ();
1003
1004           CConvert ();
1005
1006           CConvert ();
1007
1008           CConvert ();
1009
1010           CConvert ();
1011
1012           CConvert ();
1013
1014           CConvert ();
1015
1016           CConvert ();
1017
1018           CConvert ();
1019
1020           CConvert ();
1021
1022           CConvert ();
1023
1024           CConvert ();
1025
1026           CConvert ();
1027
1028           CConvert ();
1029
1030           CConvert ();
1031
1032           CConvert ();
1033
1034           CConvert ();
1035
1036           CConvert ();
1037
1038           CConvert ();
1039
1040           CConvert ();
1041
1042           CConvert ();
1043
1044           CConvert ();
1045
1046           CConvert ();
1047
1048           CConvert ();
1049
1050           CConvert ();
1051
1052           CConvert ();
1053
1054           CConvert ();
1055
1056           CConvert ();
1057
1058           CConvert ();
1059
1060           CConvert ();
1061
1062           CConvert ();
1063
1064           CConvert ();
1065
1066           CConvert ();
1067
1068           CConvert ();
1069
1070           CConvert ();
1071
1072           CConvert ();
1073
1074           CConvert ();
1075
1076           CConvert ();
1077
1078           CConvert ();
1079
1080           CConvert ();
1081
1082           CConvert ();
1083
1084           CConvert ();
1085
1086           CConvert ();
1087
1088           CConvert ();
1089
1090           CConvert ();
1091
1092           CConvert ();
1093
1094           CConvert ();
1095
1096           CConvert ();
1097
1098           CConvert ();
1099
1100           CConvert ();
1101
1102           CConvert ();
1103
1104           CConvert ();
1105
1106           CConvert ();
1107
1108           CConvert ();
1109
1110           CConvert ();
1111
1112           CConvert ();
1113
1114           CConvert ();
1115
1116           CConvert ();
1117
1118           CConvert ();
1119
1120           CConvert ();
1121
1122           CConvert ();
1123
1124           CConvert ();
1125
1126           CConvert ();
1127
1128           CConvert ();
1129
1130           CConvert ();
1131
1132           CConvert ();
1133
1134           CConvert ();
1135
1136           CConvert ();
1137
1138           CConvert ();
1139
1140           CConvert ();
1141
1142           CConvert ();
1143
1144           CConvert ();
1145
1146           CConvert ();
1147
1148           CConvert ();
1149
1150           CConvert ();
1151
1152           CConvert ();
1153
1154           CConvert ();
1155
1156           CConvert ();
1157
1158           CConvert ();
1159
1160           CConvert ();
1161
1162           CConvert ();
1163
1164           CConvert ();
1165
1166           CConvert ();
1167
1168           CConvert ();
1169
1170           CConvert ();
1171
1172           CConvert ();
1173
1174           CConvert ();
1175
1176           CConvert ();
1177
1178           CConvert ();
1179
1180           CConvert ();
1181
1182           CConvert ();
1183
1184           CConvert ();
1185
1186           CConvert ();
1187
1188           CConvert ();
1189
1190           CConvert ();
1191
1192           CConvert ();
1193
1194           CConvert ();
1195
1196           CConvert ();
1197
1198           CConvert ();
1199
1200           CConvert ();
1201
1202           CConvert ();
1203
1204           CConvert ();
1205
1206           CConvert ();
1207
1208           CConvert ();
1209
1210           CConvert ();
1211
1212           CConvert ();
1213
1214           CConvert ();
1215
1216           CConvert ();
1217
1218           CConvert ();
1219
1220           CConvert ();
1221
1222           CConvert ();
1223
1224           CConvert ();
1225
1226           CConvert ();
1227
1228           CConvert ();
1229
1230           CConvert ();
1231
1232           CConvert ();
1233
1234           CConvert ();
1235
1236           CConvert ();
1237
1238           CConvert ();
1239
1240           CConvert ();
1241
1242           CConvert ();
1243
1244           CConvert ();
1245
1246           CConvert ();
1247
1248           CConvert ();
1249
1250           CConvert ();
1251
1252           CConvert ();
1253
1254           CConvert ();
1255
1256           CConvert ();
1257
1258           CConvert ();
1259
1260           CConvert ();
1261
1262           CConvert ();
1263
1264           CConvert ();
1265
1266           CConvert ();
1267
1268           CConvert ();
1269
1270           CConvert ();
1271
1272           CConvert ();
1273
1274           CConvert ();
1275
1276           CConvert ();
1277
1278           CConvert ();
1279
1280           CConvert ();
1281
1282           CConvert ();
1283
1284           CConvert ();
1285
1286           CConvert ();
1287
1288           CConvert ();
1289
1290           CConvert ();
1291
1292           CConvert ();
1293
1294           CConvert ();
1295
1296           CConvert ();
1297
1298           CConvert ();
1299
1300           CConvert ();
1301
1302           CConvert ();
1303
1304           CConvert ();
1305
1306           CConvert ();
1307
1308           CConvert ();
1309
1310           CConvert ();
1311
1312           CConvert ();
1313
1314           CConvert ();
1315
1316           CConvert ();
1317
1318           CConvert ();
1319
1320           CConvert ();
1321
1322           CConvert ();
1323
1324           CConvert ();
1325
1326           CConvert ();
1327
1328           CConvert ();
1329
1330           CConvert ();
1331
1332           CConvert ();
1333
1334           CConvert ();
1335
1336           CConvert ();
1337
1338           CConvert ();
1339
1340           CConvert ();
1341
1342           CConvert ();
1343
1344           CConvert ();
1345
1346           CConvert ();
1347
1348           CConvert ();
1349
1350           CConvert ();
1351
1352           CConvert ();
1353
1354           CConvert ();
1355
1356           CConvert ();
1357
1358           CConvert ();
1359
1360           CConvert ();
1361
1362           CConvert ();
1363
1364           CConvert ();
1365
1366           CConvert ();
1367
1368           CConvert ();
1369
1370           CConvert ();
1371
1372           CConvert ();
1373
1374           CConvert ();
1375
1376           CConvert ();
1377
1378           CConvert ();
1379
1380           CConvert ();
1381
1382           CConvert ();
1383
1384           CConvert ();
1385
1386           CConvert ();
1387
1388           CConvert ();
1389
1390           CConvert ();
1391
1392           CConvert ();
1393
1394           CConvert ();
1395
1396           CConvert ();
1397
1398           CConvert ();
1399
1400           CConvert ();
1401
1402           CConvert ();
1403
1404           CConvert ();
1405
1406           CConvert ();
1407
1408           CConvert ();
1409
1410           CConvert ();
1411
1412           CConvert ();
1413
1414           CConvert ();
1415
1416           CConvert ();
1417
1418           CConvert ();
1419
1420           CConvert ();
1421
1422           CConvert ();
1423
1424           CConvert ();
1425
1426           CConvert ();
1427
1428           CConvert ();
1429
1430           CConvert ();
1431
1432           CConvert ();
1433
1434           CConvert ();
1435
1436           CConvert ();
1437
1438           CConvert ();
1439
1440           CConvert ();
1441
1442           CConvert ();
1443
1444           CConvert ();
1445
1446           CConvert ();
1447
1448           CConvert ();
1449
1450           CConvert ();
1451
1452           CConvert ();
1453
1454           CConvert ();
1455
1456           CConvert ();
1457
1458           CConvert ();
1459
1460           CConvert ();
1461
1462           CConvert ();
1463
1464           CConvert ();
1465
1466           CConvert ();
1467
1468           CConvert ();
1469
1470           CConvert ();
1471
1472           CConvert ();
1473
1474           CConvert ();
1475
1476           CConvert ();
1477
1478           CConvert ();
1479
1480           CConvert ();
1481
1482           CConvert ();
1483
1484           CConvert ();
1485
1486           CConvert ();
1487
1488           CConvert ();
1489
1490           CConvert ();
1491
1492           CConvert ();
1493
1494           CConvert ();
1495
1496           CConvert ();
1497
1498           CConvert ();
1499
1500           CConvert ();
1501
1502           CConvert ();
1503
1504           CConvert ();
1505
1506           CConvert ();
1507
1508           CConvert ();
1509
1510           CConvert ();
1511
1512           CConvert ();
1513
1514           CConvert ();
1515
1516           CConvert ();
1517
1518           CConvert ();
1519
1520           CConvert ();
1521
1522           CConvert ();
1523
1524           CConvert ();
1525
1526           CConvert ();
1527
1528           CConvert ();
1529
1530           CConvert ();
1531
1532           CConvert ();
1533
1534           CConvert ();
1535
1536           CConvert ();
1537
1538           CConvert ();
1539
1540           CConvert ();
1541
1542           CConvert ();
1543
1544           CConvert ();
1545
1546           CConvert ();
1547
1548           CConvert ();
1549
1550           CConvert ();
1551
1552           CConvert ();
1553
1554           CConvert ();
1555
1556           CConvert ();
1557
1558           CConvert ();
1559
1560           CConvert ();
1561
1562           CConvert ();
1563
1564           CConvert ();
1565
1566           CConvert ();
1567
1568           CConvert ();
1569
1570           CConvert ();
1571
1572           CConvert ();
1573
1574           CConvert ();
1575
1576           CConvert ();
1577
1578           CConvert ();
1579
1580           CConvert ();
1581
1582           CConvert ();
1583
1584           CConvert ();
1585
1586           CConvert ();
1587
1588           CConvert ();
1589
1590           CConvert ();
1591
1592           CConvert ();
1593
1594           CConvert ();
1595
1596           CConvert ();
1597
1598           CConvert ();
1599
1600           CConvert ();
1601
1602           CConvert ();
1603
1604           CConvert ();
1605
1606           CConvert ();
1607
1608           CConvert ();
1609
1610           CConvert ();
1611
1612           CConvert ();
1613
1614           CConvert ();
1615
1616           CConvert ();
1617
1618           CConvert ();
1619
1620           CConvert ();
1621
1622           CConvert ();
1623
1624           CConvert ();
1625
1626           CConvert ();
1627
1628           CConvert ();
1629
1630           CConvert ();
1631
1632           CConvert ();
1633
1634           CConvert ();
1635
1636           CConvert ();
1637
1638           CConvert ();
1639
1640           CConvert ();
1641
1642           CConvert ();
1643
1644           CConvert ();
1645
1646           CConvert ();
1647
1648           CConvert ();
1649
1650           CConvert ();
1651
1652           CConvert ();
1653
1654           CConvert ();
1655
1656           CConvert ();
1657
1658           CConvert ();
1659
1660           CConvert ();
1661
1662           CConvert ();
1663
1664           CConvert ();
1665
1666           CConvert ();
1667
1668           CConvert ();
1669
1670           CConvert ();
1671
1672           CConvert ();
1673
1674           CConvert ();
1675
1676           CConvert ();
1677
1678           CConvert ();
1679
1680           CConvert ();
1681
1682           CConvert ();
1683
1684           CConvert ();
1685
1686           CConvert ();
1687
1688           CConvert ();
1689
1690           CConvert ();
1691
1692           CConvert ();
1693
1694           CConvert ();
1695
1696           CConvert ();
1697
1698           CConvert ();
1699
1700           CConvert ();
1701
1702           CConvert ();
1703
1704           CConvert ();
1705
1706           CConvert ();
1707
1708           CConvert ();
1709
1710           CConvert ();
1711
1712           CConvert ();
1713
1714           CConvert ();
1715
1716           CConvert ();
1717
1718           CConvert ();
1719
1720           CConvert ();
1721
1722           CConvert ();
1723
1724           CConvert ();
1725
1726           CConvert ();
1727
1728           CConvert ();
1729
1730           CConvert ();
1731
1732           CConvert ();
1733
1734           CConvert ();
1735
1736           CConvert ();
1737
1738           CConvert ();
1739
1740           CConvert ();
1741
1742           CConvert ();
1743
1744           CConvert ();
1745
1746           CConvert ();
1747
1748           CConvert ();
1749
1750           CConvert ();
1751
1752           CConvert ();
1753
1754           CConvert ();
1755
1756           CConvert ();
1757
1758           CConvert ();
1759
1760           CConvert ();
1761
1762           CConvert ();
1763
1764           CConvert ();
1765
1766           CConvert ();
1767
1768           CConvert ();
1769
1770           CConvert ();
1771
1772           CConvert ();
1773
1774           CConvert ();
1775
1776           CConvert ();
1777
1778           CConvert ();
1779
1780           CConvert ();
1781
1782           CConvert ();
1783
1784           CConvert ();
1785
1786           CConvert ();
1787
1788           CConvert ();
1789
1790           CConvert ();
1791
1792           CConvert ();
1793
1794           C
```

```

62         virtual ~CConvert() {}
63         double GetValue (const CMaterialUSC&, enum PROPERTY);
64     };
65     CMaterialUSC ();
66     ~CMaterialUSC ();
67     CMaterialUSC (const std::string& szName, double dMassDensity,
68                   double dYieldStrength, double dModulusOfElasticity);
69 private:
70     static double m_dVConvFactor[LASTONE];
71 };
72
73 #endif

```

Lines 5 through 33 define the base class. There are four member variables – name of the material, mass density, yield strength and modulus of elasticity. The derived classes have a nested class in them – CConvert that has a member function GetValue. The GetValue function is designed to convert the value from one set of units to another. For example, the GetValue function defined within the CMaterialSI::CConvert class is designed to convert the values from SI to USC units. The static member variable m_dVConvFactor stores the conversion factor from SI to USC units for each one of the material properties. A member variable with the same name is also defined in the CMaterialUSC::CConvert. Since the member variable is static, the vector is initialized at the top of material.cpp file as

```

double CMaterialSI::m_dVConvFactor[LASTONE] = {0.00194032, 0.0208865, 0.0208865};
double CMaterialUSC::m_dVConvFactor[LASTONE] = {515.379, 47.8779, 47.8779};

```

main.cpp

```

1 #include <iostream>
2 #include "material.h"
3
4 int main ()
5 {
6     // define steel (SI units)
7     std::string szName ("Steel-0.2%C-HR");
8     CMaterialSI SteelSI (szName, 7850.0, 250.0e6, 200.0e9);
9     SteelSI.Display ();
10
11    // convert values to US Customary units
12    CMaterialSI::CConvert convertUSC;
13    CMaterialUSC SteelUSC (szName, convertUSC.GetValue (SteelSI, CMaterial::MASSDENSITY),
14                           convertUSC.GetValue (SteelSI, CMaterial::YIELDSTRENGTH),
15                           convertUSC.GetValue (SteelSI, CMaterial::MODULUSOFELASTICITY));
16    CMaterialUSC::CConvert convertSI;
17
18    // convert values back to SI and clone a new SI object
19    CMaterialSI SteelSIClone (szName, convertSI.GetValue (SteelUSC, CMaterial::MASSDENSITY),
20                             convertSI.GetValue (SteelUSC, CMaterial::YIELDSTRENGTH),
21                             convertSI.GetValue (SteelUSC, CMaterial::MODULUSOFELASTICITY));
22    // make sure we get back original values
23    std::cout << std::endl;
24    SteelSIClone.Display ();
25
26    return 0;
27 }

```

A CMaterialSI object (for widely used steel material) is defined in line 8. The current values are displayed in line 9. An object to facilitate the conversion of units is defined in line 12. Note how the scope operator :: is used as well as the fact that the definition is possible since the nested class is declared public. Another object storing values for steel in USC units is defined in lines 13-15. Note how the object to convert the units is used in conjunction with the GetValue member function. The reverse process is then defined in lines 16 through 21 to now define a clone of the original (steel)

object. We check our program by making sure that execution of the `Display` function in lines 9 and 24 displays the same values.

13.4 Function Pointers and Functors

Function pointers are variables that point to the address of a function. In other words, they are like other pointer variables except that they point to the memory address where a function is stored in memory. We first looked at function pointers in Chapter 6 (Section 6.4). Here are two examples that show how function pointers are defined and declared.

```
void (*ptQuadraticRoots) (float a, float b, float c, float& r1, float& r2);
int (CMymath::*ptLinearInterp) (float m, float c, float x, float& fValue);
```

Note that the names of the variables in the two examples are `ptQuadraticRoots` and `ptLinearInterp`. Once the function pointer variables are declared, they can be used just like any other variable.

Example Program 13.4.1

In this example we will illustrate how non-class member pointer variables can be used.

main.cpp

```
1 #include <iostream>
2
3 float AddSimpleTwo (float fA, float fB) { return (fA+fB); }
4 float AddSquareTwo (float fA, float fB) { return (fA*fA+fB*fB); }
5
6 void Use_In_A_Function (float (*ptrFunc)(float fA, float fB))
7 {
8     // use with comparison operators
9     if (!ptrFunc)
10         std::cout << "Add function is not initialized.\n";
11     else
12     {
13         if (ptrFunc == &AddSimpleTwo)
14             std::cout << "Variable points to function AddSimpleTwo\n";
15         else if (ptrFunc == &AddSquareTwo)
16             std::cout << "Variable points to function AddSquareTwo\n";
17     }
18     std::cout << "Function call gives "
19             << ptrFunc(2.3f, 3.4f) << '\n';
20 }
21
22 int main ()
23 {
24     // define a pointer variable to a function
25     // with a specified signature
26     float (*ptAddFunction) (float, float) = NULL;
27
28     // assign the address of the function
29     ptAddFunction = &AddSimpleTwo;
30     // use the function
31     std::cout << "Simple addition gives "
32             << ptAddFunction(2.3f, 3.4f) << '\n';
33
34     // pass the pointer variable as an argument
35     Use_In_A_Function (ptAddFunction);
36
37     return 0;
38 }
```

In lines 3 and 4 we define two simple functions completely. Both these functions return a float value and accept two float variables as function arguments. However, their names and functionalities are completely different. In lines 6 through 20 we define a function that has a single argument – a pointer to a function that returns a float value and accepts two float variables as the function arguments.

We start illustrating the usage of the function pointer variables in the main program. In line 26 we define a variable called `ptAddFunction` that is a pointer to a function that returns a float value and accepts two float variables as function arguments. The variable is initialized to `NULL`. In line 29, the variable is given a value, the address of the function `AddSimpleTwo`. The function is invoked in line 32 with the arguments are 2.3 and 3.4.

How the variable can be used as an argument in a function call is shown in line 35 where the function `Use_In_A_Function` is called. More function pointer usage is shown in that function. In line 9, the `!` logical operator is used to check if the variable is initialized with an appropriate address. In lines 13 and 15 the `==` comparison operator is used to check what function is passed as the argument. Finally, line 19 shows how the variable is used to invoke the function similar to the usage in line 32.

Example Program 13.4.2

In this example we will illustrate how class member pointer variables can be used using essentially the same example as in 13.4.1. Three source code files are used. The class containing the two add functions is called `CMyAddLibrary`. They are shown first followed by the main program.

`myaddlibrary.h`

```

1  #ifndef __RAJAN__MYADDLIBRARY_H__
2  #define __RAJAN__MYADDLIBRARY_H__
3
4  class CMyAddLibrary
5  {
6      public:
7          CMyAddLibrary ();
8          ~CMyAddLibrary ();
9
10     float AddSimpleTwo (float, float);
11     float AddSquareTwo (float, float);
12     void Use_In_A_Function (float (CMyAddLibrary::*ptrFunc)(float fA, float fB));
13
14     private:
15 };
16
17 #endif

```

The major difference in the declaration of the member function `Use_In_A_Function` is the use of the scope operator `::` to qualify the the member pointer variable is associated with the `CMyAddLibrary` class (line 12).

`myaddlibrary.cpp`

```

1  #include <iostream>
2  #include "myaddlibrary.h"
3
4  CMyAddLibrary::CMyAddLibrary ()
5  {
6
7
8  CMyAddLibrary::~CMyAddLibrary ()
9  {

```

```

10 }
11
12 float CMyAddLibrary::AddSimpleTwo (float fA, float fB)
13 {
14     return (fA + fB);
15 }
16
17 float CMyAddLibrary::AddSquareTwo (float fA, float fB)
18 {
19     return (fA*fA + fB*fB);
20 }
21
22 void CMyAddLibrary::Use_In_A_Function (float (CMyAddLibrary::*ptrFunc)(float fA, float fB))
23 {
24     // use with comparison operators
25     if (!ptrFunc)
26         std::cout << "Add function is not initialized.\n";
27     else
28     {
29         if (ptrFunc == &CMyAddLibrary::AddSimpleTwo)
30             std::cout << "Variable points to member function AddSimpleTwo\n";
31         else if (ptrFunc == &CMyAddLibrary::AddSquareTwo)
32             std::cout << "Variable points to member function AddSquareTwo\n";
33     }
34     std::cout << "Function call gives "
35     << (*this.*ptrFunc)(2.3f, 3.4f) << '\n';
36 }

```

Once again we point the major differences in the definition and the usage. Lines 29 and 31 must now contain the class name (`CMyAddLibrary::`) so that the scope of the usage is clear. In line 35, the appropriate member function from the class can be invoked only by using (`*this.*ptrFunc`) qualifier.

main.cpp

```

1 #include <iostream>
2 #include "myaddlibrary.h"
3
4 int main ()
5 {
6     // define a pointer variable to a function
7     // with a specified signature
8     float (CMyAddLibrary::*ptAddFunction) (float, float) = NULL;
9     CMyAddLibrary MALL;
10
11    // assign the address of the function
12    ptAddFunction = &CMyAddLibrary::AddSimpleTwo;
13    // use the function
14    std::cout << "Simple addition gives "
15    << (MALL.*ptAddFunction)(2.3f, 3.4f) << '\n';
16
17    // pass the pointer variable as an argument
18    MALL.Use_In_A_Function (ptAddFunction);
19
20    // reassign and reuse
21    ptAddFunction = &CMyAddLibrary::AddSquareTwo;
22    MALL.Use_In_A_Function (ptAddFunction);
23
24    return 0;
25 }

```

There are several changes to bring in the class (object-oriented) associated usage. As before, we declare a pointer variable to a function in line 8; however we need to add the class qualifier (`CMyAddLibrary::`). In addition, we need an object associated with the class – this is specified in line 9. Note that lines 12, 15 and 18 are essentially the same as before except that we now need the class

qualifier. In lines 21 and 22 we show how the variable value can be reassigned and how the variable can be reused.

Functors

Function pointers can be used in the context of callback functions. A callback function is a function that is called using a function pointer. We have already seen callback function usage not only in this section but also in Chapter 6. A Function Object, or Functor is simply any object that can be called as if it is a function. Functors can encapsulate function-pointers in C and C++ using templates and polymorphism. Thus you can build up a list of pointers to member-functions of arbitrary classes and call them all through the same interface without bothering about their class or the need of a pointer to an instance⁴. However, all the functions must have the same return-type and calling parameters. Sometimes Functors are also known as Closures. Functors can also be used to implement callbacks.

We start the implementation by first defining a base class `FunctorABC` that provides a virtually overloaded operator () with which one will be able to call the required member function. From the base class we derive a template class that is initialized with a pointer to an object and a pointer to a member function in its constructor. The derived class overrides the operator () of the base class. In the overridden versions it calls the member function using the stored pointers to the object and to the member function. Here is an example that illustrates the ideas.

Example Program 13.4.3

We will create our own math library that has two functions that (1) accept 2 numbers as input (say, a and b) and return a value, and (2) supports the following computations using a and b as (i) function `Simple2` that computes $(a+b)$, and (ii) function `Square2` that computes $(a^2 + b^2)$.

We first start by defining an abstract base class `FunctorABC` that provides a virtually overloaded operator () with which one will be able to call the required member function.

`functorABC.h`

```

1  #ifndef __RAJAN_FUNCTORABC_H__
2  #define __RAJAN_FUNCTORABC_H__
3
4  // abstract base class
5  template <class Type> class FunctorABC
6  {
7      public:
8          // operator () overloading
9          // virtual since derived classes will use a pointer to an object
10         // and a pointer to a member function to make the function call
11         virtual Type operator()(Type A, Type B) = 0;
12     };
13
14 #endif

```

⁴ Lars Haendel, “Introduction to C/C++ Function-Pointers, Callbacks and Functors”, www.function-pointer.org.

Note that the class is a template class with Type being int, float, double etc. The () operator is overloaded and is a pure virtual function.

Next we define our math library.

math2lib.h

```

1  #ifndef _RAJAN_MATH2LIB_H_
2  #define _RAJAN_MATH2LIB_H_
3
4  #include "FunctorABC.h"
5
6  // derived template class
7  template <class TClass, class Type> class CMath2Lib : public FunctorABC<Type>
8  {
9      public:
10         // constructor - takes pointer to an object and pointer to a member
11         // and stores them in two private variables
12         CMath2Lib (TClass* _pt2Object, Type(TClass::*_fpt)(Type, Type))
13         { m_pt2Object = _pt2Object; m_fpt=_fpt; }
14
15         // override operator "()"
16         virtual Type operator()(Type a, Type b)
17         { return (*m_pt2Object.*m_fpt)(a, b); }
18
19     private:
20         Type (TClass::*m_fpt)(Type a, Type b); // pointer to member function
21         TClass* m_pt2Object; // pointer to object
22     };
23
24 // specific functions in the library follow
25 template <class Type> class CAddSimple2
26 {
27     public:
28         CAddSimple2(){};
29         Type AddThem (Type a, Type b) { return (a+b); };
30     };
31
32 template <class Type> class CAddSquare2
33 {
34     public:
35         CAddSquare2(){};
36         Type AddThem (Type a, Type b) { return (a*a + b*b); };
37     };
38
39 #endif

```

Our math library is a template class derived from the abstract base class, FunctorABC. The class object is invoked (see line 7) by specifying one of the math functions in the library and the data type associated with the computations, and the constructor (lines 12 and 13) takes a pointer to an object associated with one of the math functions and a pointer to the member function associated with the math function that actually carries out the computation. For example

```
CMath2Lib<MathFunction, DataType> object(&MathFunctionObject, &MemberFunction);
```

Or CMath2Lib<CAddSimple2<int>, int> AS2(&CAddSimple2<int>, &CAddSimple2<int>::AddThem);

The two functions in the library are defined in lines 25 though 37. Each one has a public member function AddThem that implements the actual computation associated with the function. These member functions are never called directly. They are invoked via the overloaded operator (lines 16 and 17).

main.cpp

```

1  #include <iostream>
2  #include "Math2Lib.h"
3
4  int main ()
5  {
6      // 1. instantiate objects (specific math functions)
7      CAddSimple2<float> OSimple2;
8      CAddSquare2<float> OSquare2;
9
10     // 2. instantiate objects ...
11     //   functor which encapsulates pointer to object
12     //   and to specific member function
13     CMath2Lib<CAddSimple2<float>, float>
14         simple2(&OSimple2, &CAddSimple2<float>::AddThem);
15     CMath2Lib<CAddSquare2<float>, float>
16         square2(&OSquare2, &CAddSquare2<float>::AddThem);
17
18     // 3. make array with pointers to FunctorABC, the base class,
19     //   and initialize it
20     FunctorABC<float>* TOF[] = {&simple2, &square2};
21     enum {SIMPLE=0, SQUARE};
22
23     // 4. use array to call member functions without the need for an object
24     std::cout << "AddSimple value : "
25             << (*TOF[SIMPLE]) (2.3f, 3.4f) << '\n';
26     std::cout << "AddSquare value : "
27             << (*TOF[SQUARE]) (2.3f, 3.4f) << '\n';
28
29     return 0;
30 }
```

The overall process of using the functors is a 4 step process with the first 3 steps being used to set up the usage and step 4 being the actual use. In the first step we instantiate objects associated with the two functions (lines 7 and 8) in the math library. The next step involves instantiating the objects, the functors that will be used in invoking the two functions. To make this invocation simple, we need the third step where we define the abstract base class array where each element in the array contains the address of the pointer objects that can be used to invoke the individual functions in the library. From this point onwards, the math functions can be invoked simply by using the elements of this array as seen in lines 25 and 27.

Summary

This chapter completes the formal treatment of C++ related object-oriented techniques in this book. In the rest of the chapters, we will see how to use these OO ideas to develop and implement algorithms and solve engineering and scientific problems numerically.

EXERCISES

Appetizers

Problem 13.1

TBD

Problem 13.2

TBD

Main Course

Problem 13.3

TBD

Problem 13.4

Develop a template class (`COOMatrix`) to store two-dimensional arrays similar to the `CMatrix` class that we saw in Chapter 9. However, derive this class from the `CVector` class and store the elements of the matrix in a vector. Note that if we have a matrix $A_{m \times n}$, then element A_{ij} is located at $(i-1)n + j$ if the matrix is stored row-wise and at $(j-1)m + i$ if the matrix is stored column-wise. Write a few test programs to compare the runtime performance of this implementation with the `CMatrix` implementation.

C++ Concepts

Problem 13.5

TBD

Chapter

14

Ordinary Differential Equations

"The person who knows "how" will always have a job. The person who knows "why" will always be his boss." Diane Ravitch

'Real learning comes about when the competitive spirit has ceased.' J. Krishnamurti

In earlier courses in Statics and Deformable Solids (or, Strength of Materials), most of the structural systems were statically determinate.

Objectives

- To understand the syntax of C++ programs.
- To understand the concepts associated with data types, variables, arithmetic expressions, assignment statements and simple input and output.
- To understand and practice writing C++ programs.

14.1 Ordinary Differential Equations

Below we present a complete C++ program.

14.2 Case Study: A One-Dimensional ODE Toolbox

EXERCISES

Appetizers

Main Course

Numerical Analysis Concepts

Partial Differential Equations

"The central enemy of reliability is complexity" Geer et al.

"Measuring programs by counting the lines of code is like measuring aircraft quality by weight."

Anon

"The purpose of computing is insight, NOT numbers."

Richard Hamming

Objectives

- To understand the basic concepts associated with partial differential equations.
- To understand the different engineering examples of PDEs.
- To understand the basics of finite element method (FEM) and specifically, the Galerkin's Method.
- To understand and practice numerical solution of one-dimensional PDEs via the finite element method.

15.1 Background

One-dimensional boundary-value problems influence a variety of engineering areas - we list some of the more popular examples below.

Specialty Area	Problem Description
Solid Mechanics	Transverse deflection of a cable
Hydrodynamics	One-dimensional flow in an inviscid, incompressible fluid
Magnetostatics	One-dimensional magnetic potential distribution
Heat Conduction	One-dimensional heat flow in a solid medium
Electrostatics	One-dimensional electric potential distribution

Consider a one-dimensional boundary value problem (also known as equilibrium problem) given by

$$\frac{d^2y}{dx^2} + P(x)\frac{dy}{dx} + Q(x)y = F(x) \quad (15.1.1)$$

If P and Q are constants, then

$$\frac{d^2y}{dx^2} + P\frac{dy}{dx} + Qy = F(x) \quad (15.1.2)$$

The total solution is

$$y(x) = Ae^{\lambda_1 x} + Be^{\lambda_2 x} + C_0F(x) + C_1F'(x) + \dots \quad (15.1.3)$$

where the constants of integration A and B can be found by substituting the two boundary conditions into Eqn. (15.1.2). Note that we need two boundary conditions for the problem to be well-posed.

The boundary conditions are of three types.

- (a) The function y may be specified. This is known as the Dirichlet or Essential boundary condition.
- (b) The derivative y' may be specified. This is known as the Neumann or Natural boundary condition.
- (c) The function y and the derivative y' may be specified. This is known as the Mixed or Robin boundary condition.

In the FE solution, an approximate or trial solution $\tilde{y}(x)$ is constructed and solved for¹. The FE approach has three distinct operations.

- (a) A trial solution $\tilde{y}(x)$ is constructed.
- (b) An optimizing criterion is applied to $\tilde{y}(x)$.
- (c) An estimation of the accuracy of $\tilde{y}(x)$ is made.

Trial Solution

The trial solution is constructed with a finite number of terms as

$$\tilde{y}(x; a) = \phi_0(x) + a_1\phi_1(x) + a_2\phi_2(x) + \dots + a_n\phi_n(x) \quad (15.1.4)$$

where $\phi_i(x)$ are known trial or basis functions and the coefficients a_i are undetermined parameters known as degrees of freedom (DOF). The purpose of the trial function $\phi_0(x)$ is to satisfy some or all of the boundary conditions. The most common form of trial solutions is to use polynomials. We will see more about this later.

Optimizing Criterion

The optimizing criterion is used to generate the appropriate equations so that we can solve for the numerical values of the coefficients a_i . As you can guess, the optimizing criterion is not unique and different approaches define what is meant by the “best possible approximation” to the exact solution. The two most common forms are

- (a) The Method of Weighted Residuals (MWR) – applicable when the problem is described by differential equations, and
- (b) The Ritz Variational Method (RVM) - applicable when the problem is described by integral (or, variational) equations.

In this lesson, the focus is on the former method. In the MWR, the criteria minimize an expression of error in the differential equation. In the RVM, an attempt is made to extremize (typically, minimize) a physical quantity. We will see this approach in the second module of this course.

¹ Those familiar with the finite difference solution will note that there are two approaches to solving the boundary-value ODE – the shooting (initial-value) method and the equilibrium (boundary-value) method.

Method of Weighted Residuals

There are at least four different optimizing criteria and we will see them in this lesson. Consider the differential equation (15.1.1) rewritten as

$$\frac{d^2y}{dx^2} + P(x)\frac{dy}{dx} + Q(x)y - F(x) = 0 \quad (15.1.5a)$$

or, $\mathfrak{J}(y) - F(x) = 0 \quad (15.1.5b)$

Substituting the trial solution, we have, in general

$$R(x; a) = \tilde{\mathfrak{J}}(y) - F(x) \neq 0 \quad (15.1.6)$$

$R(x; a)$ is known as the residual or error in the solution. In the MWR, the optimizing criterion is to find the numerical values for a_i which will make $R(x; a)$ as close to zero as possible for all values of x throughout the domain of the problem. Once a specific criterion is applied, a set of algebraic equations is produced. As you can see, the process is to transform the original (linear) ODE to a set of linear algebraic equations.

Collocation Method

In this method, for each of the parameter a_i a point x_i is chosen and the residual is set to zero at that point.

$$R(x_i; a) = 0 \quad i = 1, \dots, n \quad (15.1.7)$$

The points x_i are known as the collocation points. Note that we are setting the error in the residual, not the error in the solution, to zero.

Subdomain Method

In this method, for each of the parameter a_i an interval Δx_i is chosen and the average of the residual is set to zero.

$$\frac{1}{\Delta x_i} \int_{\Delta x_i} R(x; a) dx = 0 \quad i = 1, \dots, n \quad (15.1.8)$$

The intervals Δx_i are called the subdomains.

Least-squares Method

In this method, with respect to each a_i we minimize the integral over the entire domain, the square of the residual.

$$\frac{\partial}{\partial a_i} \int_{\Omega} R^2(x; a) dx = 0 \quad i = 1, \dots, n \quad (15.1.9)$$

The Galerkin's Method

In this method, for each a_i we require that a weighted average of the residual over the entire domain be zero. The weighting functions are the trial functions $\phi_i(x)$ associated with each a_i .

$$\int_{\Omega} R(x; a) \phi_i(x) dx = 0 \quad i = 1, \dots, n \quad (15.1.10)$$

The natural question is “Which of these techniques is the most appropriate?” A detailed answer is outside the scope of this text. However, experience has shown that the Galerkin's Method is the most suitable for the type of finite element applications discussed in this text.

Example 15.1.1

Let us look at an example to see how the Galerkin's Method works. Consider the problem

$$\text{DE: } \frac{d}{dx} \left(x \frac{dy}{dx} \right) = \frac{2}{x^2} \quad 1 \leq x \leq 2 \quad (15.1.11)$$

$$\text{BC: } y(x=1) = 2 \quad (15.1.12)$$

$$\left(-x \frac{dy}{dx} \right)_{x=2} = \frac{1}{2} \quad (15.1.13)$$

Classical (or, Theoretical) Solution

Let us assume the trial solution as

$$\tilde{y} = a_1 + a_2 x + a_3 x^2 + a_4 x^3 \quad (15.1.14a)$$

Hence,

$$\frac{d \tilde{y}}{dx} = a_2 + 2a_3 x + 3a_4 x^2 \quad (15.1.14b)$$

In order to satisfy the BC, we need

$$\tilde{y}(x=1) = 2 = a_1 + a_2(1) + a_3(1)^2 + a_4(1)^3$$

$$\text{or, } a_1 + a_2 + a_3 + a_4 = 2 \quad (15.1.15)$$

And,

$$\left(-x \frac{d\tilde{y}}{dx} \right)_{x=2} = \frac{1}{2} = -2a_2 - 8a_3 - 24a_4$$

$$\text{or, } a_2 + 4a_3 + 12a_4 = -\frac{1}{4} \quad (15.1.16)$$

Eqns. (15.1.15) and (15.1.16) are known as constraint equations since the 4 a_i 's in Eqn. (15.1.14a) are no longer independent. The two constraint equations render only 2 out of the 4 a_i 's as independent parameters (or, DOF). Using the two constraint equations to write a_1 and a_2 in terms of the other two, we have

$$a_1 = 2 - a_2 - a_3 - a_4 \quad (15.1.17)$$

$$a_2 = -\frac{1}{4} - 4a_3 - 12a_4 \quad (15.1.18)$$

Substituting these two equations in (15.1.14a)

$$\tilde{y} = 2 - \frac{1}{4}(x-1) + a_3(x-1)(x-3) + a_4(x-1)(x^2 + x - 11) \quad (15.1.19)$$

This equation is now in the familiar form

$$\tilde{y}(x; a) = \phi_0(x) + a_1\phi_1(x) + a_2\phi_2(x) \quad (15.1.20)$$

$$\text{where } \phi_0(x) = 2 - \frac{1}{4}(x-1)$$

$$\phi_1(x) = (x-1)(x-3)$$

$$\phi_2(x) = (x-1)(x^2 + x - 11)$$

and we will assume for the sake of convenience that a_1 and a_2 in (15.2.20) are the same as a_3 and a_4 in (15.2.19).

We will now write the residual as

$$R(x; a) = \frac{d}{dx} \left(x \frac{d\tilde{y}(x)}{dx} \right) - \frac{2}{x^2} \neq 0 \quad (15.1.21)$$

Substituting (T3L2-10) in the above equation, we have

$$R(x; a) = -\frac{1}{4} + 4(x-1)a_1 + 3(3x^2 - 4)a_2 - \frac{2}{x^2} \quad (15.1.22)$$

Now using the Galerkin's Method (Eqn. (15.1.10))

$$\begin{aligned} & \int_1^2 \left(-\frac{1}{4} + 4(x-1)a_1 + 3(3x^2 - 4)a_2 - \frac{2}{x^2} \right) (x-1)(x-3) dx = 0 \\ & \int_1^2 \left(-\frac{1}{4} + 4(x-1)a_1 + 3(3x^2 - 4)a_2 - \frac{2}{x^2} \right) (x-1)(x^2 + x - 11) dx = 0 \end{aligned} \quad (15.1.23a)$$

Integrating yields two equations

$$\begin{bmatrix} \frac{5}{3} & \frac{41}{5} \\ \frac{3}{41} & \frac{5}{81} \\ \frac{41}{5} & \frac{2}{2} \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \end{Bmatrix} = \begin{Bmatrix} -\frac{29}{6} + 8\ln 2 \\ -\frac{211}{16} + 24\ln 2 \end{Bmatrix} \quad (15.1.23b)$$

And solving

$$a_1 = 2.138 \text{ and } a_2 = -0.348$$

Hence, substituting in Eqn. (15.1.10) yields

$$\begin{aligned} \tilde{y}(x; a) &= 2 - \frac{1}{4}(x-1) + 2.138(x-1)(x-3) - 0.348(x-1)(x^2 + x - 11) \\ &= -0.348x^3 + 2.138x^2 - 4.629x + 4.839 \end{aligned} \quad (15.1.24)$$

There is an important concept associated with a derived term - *flux* that we have not seen before and should have. Flux is defined as

$$\tau(x) = -x \frac{dy}{dx} \quad (15.1.25)$$

Flux has a physical interpretation and we will discuss the concepts in the next topic. Substituting in Eqn. (15.1.15), we have

$$\begin{aligned} \tilde{\tau}(x; a) &= \frac{1}{2} + \frac{1}{4}(x-2) - 4.276x(x-2) + 1.043x(x-2)(x+2) \\ &= 1.043x^3 - 4.276x^2 + 4.629x \end{aligned} \quad (15.1.26)$$

Finally, note that this solution is called the theoretical (or, classical) solution because of the manner in which the two boundary conditions were imposed. In this methodology, the BCs were imposed on the trial solution itself so that the trial solution would satisfy the BCs exactly. However, this in no way implies that the solution is correct in the interior of the problem domain.

Some Important Observations

- Why did we start with a trial solution that is a cubic polynomial? Since there are two BCs that must be imposed, the *lowest* order polynomial that we could have used would have been a quadratic polynomial. Imposing the BCs then would have left one free parameter (or, one DOF). The choice of using a cubic polynomial was an arbitrary choice and we were left with two free parameters.
- The classical way of applying the BCs can become extremely cumbersome with more complex problems.
- How do we know whether the solution is good? One way to answer this question is to look at the concept of *convergence*. The exact solution to the problem is

$$y(x) = \frac{2}{x} + \frac{1}{2} \ln x \quad (15.2.27a)$$

and

$$\tau(x) = \frac{2}{x} - \frac{1}{2} \quad (15.2.27b)$$

Convergence can be checked by starting with a low-order polynomial and increasing the order of the polynomial gradually. The different trial functions should converge to a solution. If we had started with a quadratic polynomial, the Galerkin's solution would have been

$$\begin{aligned}\tilde{y}(x; a) &= 2 - \frac{1}{4}(x-1) + 0.427(x-1)(x-3) \\ &= 0.427x^2 - 1.958x + 3.531\end{aligned}\tag{15.2.28}$$

and

$$\tilde{\tau} = \frac{1}{2} + \frac{1}{4}(x-2) - 0.854x(x-2) = -0.854x^2 + 1.958x\tag{15.2.29}$$

In the previous section, with the trial solution as a cubic polynomial the solution was

$$\begin{aligned}\tilde{y}(x; a) &= 2 - \frac{1}{4}(x-1) + 2.138(x-1)(x-3) - 0.348(x-1)(x^2 + x - 11) \\ &= -0.348x^3 + 2.138x^2 - 4.629x + 4.839\end{aligned}$$

and

$$\begin{aligned}\tilde{\tau}(x; a) &= \frac{1}{2} + \frac{1}{4}(x-2) - 4.276x(x-2) + 1.043x(x-2)(x+2) \\ &= 1.043x^3 - 4.276x^2 + 4.629x\end{aligned}$$

What if we had started with a quartic polynomial? The solution is then

$$\begin{aligned}\tilde{y}(x; a) &= 2 - \frac{1}{4}(x-1) + 3.3725(x-1)(x-3) - 0.8881(x-1)(x^2 + x - 11) + \\ &\quad + 0.0864(x-1)(x^3 + x^2 + x - 31) \\ &= 0.0864x^4 - 0.888x^3 + 3.3725x^2 - 5.848x + 5.277\end{aligned}\tag{15.2.30}$$

and

$$\begin{aligned}\tilde{\tau}(x; a) &= \frac{1}{2} + \frac{1}{4}(x-2) - 6.745x(x-2) + 2.664x(x-2)(x+2) - 0.346x(x-2)(x^2 + 2x + 4) \\ &= -0.346x^4 + 2.664x^3 - 6.745x^2 + 5.848x\end{aligned}\tag{15.2.31}$$

We will now plot the three trial functions and the exact solution both for the function and the flux.

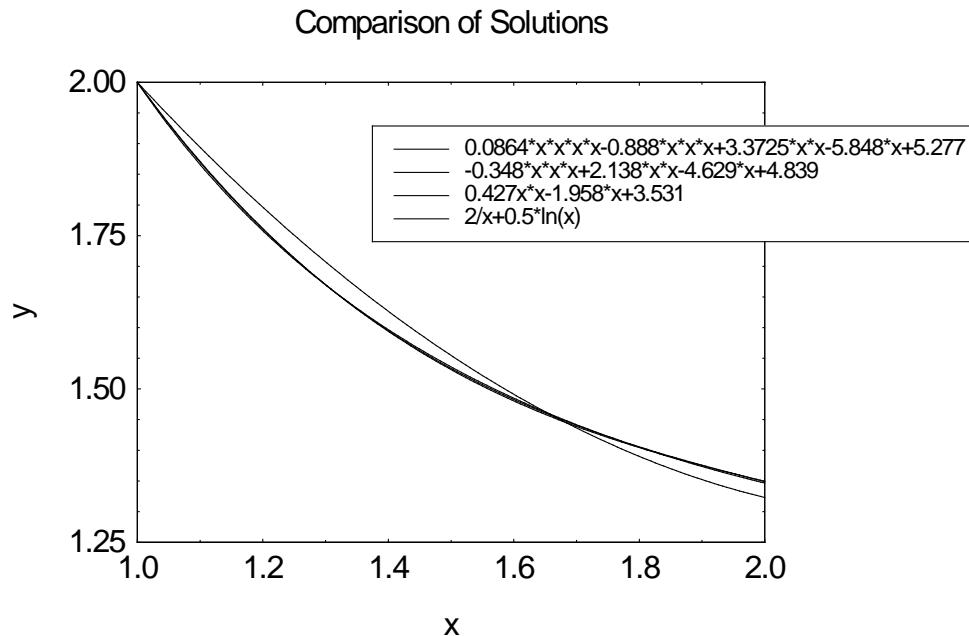


Fig. 15.1.1 Comparison of the function

While some difference can be noted between the quadratic and the exact solution, there is very little difference between the cubic, quartic and the exact solutions. Note that all the solutions have the same function value at the left boundary point $x = 1$.

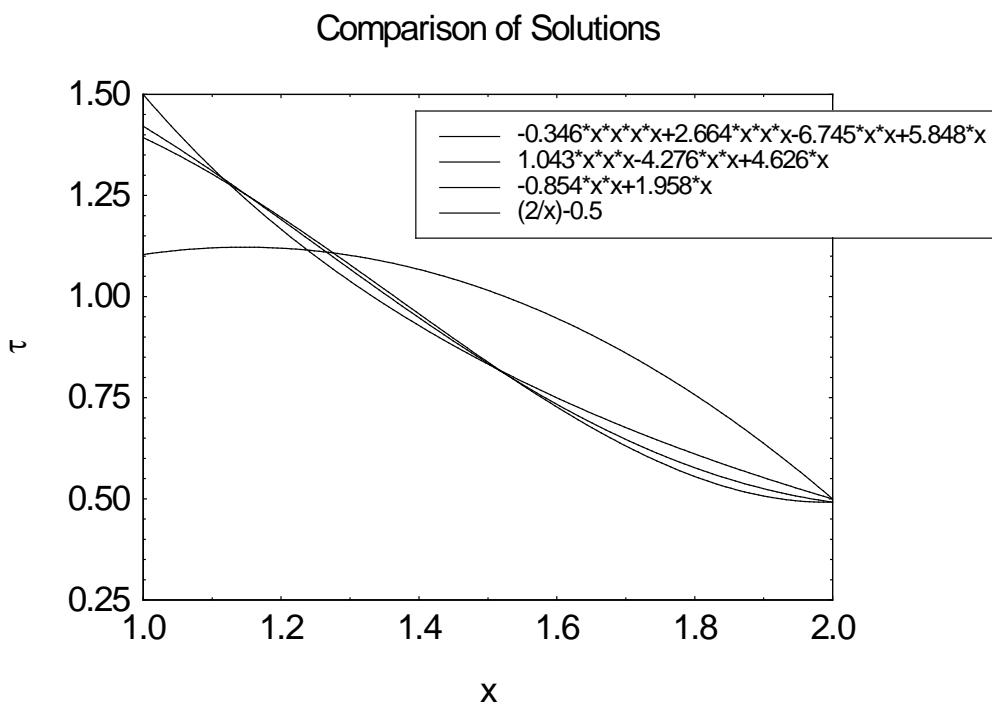


Fig. 15.1.2 Comparison of the flux

The differences between the exact solution and the three Galerkin's solutions with respect to the flux is a different matter altogether. The error in the flux from the quadratic solution is large. However, the error decreases with the cubic and the quartic functions. Note that all the solutions have the same flux value at the right boundary point $x = 2$. The message here is quite clear – (a) small errors in the function do not translate to small errors in the flux that involve the derivatives of the function, and (b) increasing order (polynomials) trial functions yield better and converging solutions².

The residual function is the function corresponding to the original differential equation (with all the terms on the LHS) in which an approximate solution is substituted. It measures how close the approximation is to satisfying the DE but does not tell us how close the approximation is to the exact solution. The Method of Weighted Residuals converts the original DE into a set of algebraic equations that are much easier to solve. There are different approaches used in terms of weighting the residual. Of all the techniques discussed, the Galerkin's Method is by far the most suitable for the type of problems encountered in engineering analysis.

In the lessons in this topic, we saw how to assume the approximate solution called the trial solution and use the Galerkin's Method to generate the algebraic equations. The trial solution is usually a polynomial because of the properties that they possess (continuous, differential, easy to handle etc.). We saw how to enforce the boundary conditions on the trial solution. We also looked at the *flux* term. Finally, we also saw how to obtain more accurate solutions by increasing the order of the polynomial in the trial solution.

There are two problems with this classical (or, theoretical approach). First, the trial solution is valid for the entire problem domain. Hence it is not possible to accurately model problems in which there are known discontinuities in the solution and the flux. Second, the manner in which the boundary conditions are enforced can become cumbersome for more complex problems. In this next topic, we will see how to overcome both these drawbacks.

² Later we will see that the trial functions must have certain properties for this to be true. If we keep on increasing order of the trial solution will we converge to the exact solution for this problem?

EXERCISES

Problem 15.1.1

Consider the differential equation

$$\frac{d^2y}{dx^2} + \frac{y}{4} = 0 \quad 0 < x < \pi$$

with $y(0) = 1$ and $y(\pi) = 0$. Find the solution using the Galerkin's Method by assuming the trial solution as $y(x; a) = a_1 + a_2x + a_3x^2 + a_4x^3$. Compare with the exact solution.

Problem 15.1.2

Consider the differential equation

$$\frac{d}{dx} \left(x^2 \frac{dy}{dx} \right) = \frac{1}{12} (-30x^4 + 204x^3 - 351x^2 + 110x) \quad 0 < x < 4$$

with $y(0) = 1$ and $y(4) = 0$. Find the solution using the Galerkin's Method by assuming the trial solution as (i) a quadratic polynomial, and (ii) a cubic polynomial. Compare to the exact solution.

Problem 15.1.3

Consider the differential equation

$$\frac{d}{dx} \left((x+1) \frac{dy(x)}{dx} \right) = 0 \quad 1 < x < 2$$

with $y(x=1) = 1 \quad \left(-(x+1) \frac{dy}{dx} \right)_{x=2} = 1$

- (a) Using the Galerkin's method with a quadratic polynomial for the trial solution obtain an approximate solution for both the function and the flux.
- (b) Obtain a second approximate solution using a cubic polynomial for the trial solution.
- (c) Compare the two solutions with each other and the exact solution.

15.2 The Element Concept

Earlier we looked at applying the Galerkin's Method in the classical (or, theoretical) sense. To make the approach general and useful we need to deviate to a new format as shown below.

Step 1: Let the trial solution be assumed as

$$\tilde{y}(x; a) = \phi_0(x) + a_1\phi_1(x) + a_2\phi_2(x) + \dots + a_n\phi_n(x) = \phi_0(x) + \sum_{i=1}^n a_i\phi_i(x) \quad (15.2.1)$$

We will stick with the example from the previous topic. Using the definition of the residual, we have

$$R(x; a) = \frac{d}{dx} \left(x \frac{d \tilde{y}(x)}{dx} \right) - \frac{2}{x^2} \quad (15.2.2)$$

Since we have n parameter trial function, we need n residual equations (see Eqn. (15.2.10))

$$\int_{x_a}^{x_b} R(x; a)\phi_i(x)dx = 0 \quad i = 1, \dots, n \quad (15.2.3)$$

Notice that we have changed the limits of integration (instead of using 1 and 2 as the limits) just to make the derivation general and more useful. Substituting, we have

$$\int_{x_a}^{x_b} \left[\frac{d}{dx} \left(x \frac{d \tilde{y}(x)}{dx} \right) - \frac{2}{x^2} \right] \phi_i(x)dx = 0 \quad i = 1, \dots, n \quad (15.2.4)$$

Step 2: Integrate by parts the highest derivative term in the residual equations (which would be first term containing the second-order derivative). Rearranging the terms, we have

$$\int_{x_a}^{x_b} x \frac{d \tilde{y}}{dx} \frac{d \phi_i}{dx} dx = - \int_{x_a}^{x_b} \frac{2}{x^2} \phi_i dx - \left[\left(-x \frac{d \tilde{y}}{dx} \right) \phi_i \right]_{x_a}^{x_b} \quad i = 1, \dots, n \quad (15.2.5)$$

Three observations here – (a) the highest order derivative in Eqn. (15.2.5) is now lower (only first order derivatives compared to second-order in Eqn. (15.2.4)), and (b) the “stiffness” term is on the LHS and the “loading” terms are on the RHS, and (c) the loading term contains the boundary flux term.

Step 3: Substituting Eqn. (15.2.1) in Eqn. (15.2.5) and noting that

$$\frac{d \tilde{y}(x; a)}{dx} = \frac{d \phi_0(x)}{dx} + \sum_{i=1}^n a_i \frac{d \phi_i(x)}{dx} \quad (15.2.6)$$

we have

$$\sum_{j=1}^n \left(\int_{x_a}^{x_b} \frac{d\phi_i}{dx} x \frac{d\phi_j}{dx} dx \right) a_j = - \int_{x_a}^{x_b} \frac{2}{x^2} \phi_i dx - \left[\left(-x \frac{dy}{dx} \right) \phi_i \right]_{x_a}^{x_b} \\ - \int_{x_a}^{x_b} \frac{d\phi_i}{dx} x \frac{d\phi_0}{dx} dx \quad i = 1, \dots, n \quad (15.2.7)$$

Let

$$K_{ij} = \int_{x_a}^{x_b} \frac{d\phi_i}{dx} x \frac{d\phi_j}{dx} dx$$

and $F_i = - \int_{x_a}^{x_b} \frac{2}{x^2} \phi_i dx - \left[\left(-x \frac{dy}{dx} \right) \phi_i \right]_{x_a}^{x_b} - \int_{x_a}^{x_b} \frac{d\phi_i}{dx} x \frac{d\phi_0}{dx} dx \quad (15.2.8)$

Then we can rewrite Eqn. (15.2.7) in the matrix notation as

$$\begin{bmatrix} K_{11} & K_{12} & \dots & K_{1n} \\ K_{21} & K_{22} & \dots & K_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ K_{n1} & K_{n2} & \dots & K_{nn} \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{Bmatrix} = \begin{Bmatrix} F_1 \\ F_2 \\ \vdots \\ F_n \end{Bmatrix} \quad (15.2.9)$$

or,

$$\mathbf{K}_{n \times n} \mathbf{a}_{n \times 1} = \mathbf{F}_{n \times 1} \quad (15.2.10)$$

The stiffness matrix \mathbf{K} is symmetric since

$$K_{ji} = \int_{x_a}^{x_b} \frac{d\phi_j}{dx} x \frac{d\phi_i}{dx} dx = K_{ij} \quad (15.2.8b)$$

Step 4: Use a specific form of the trial solution

In the previous topic we used a polynomial as the trial solution. Polynomials are easy to deal with and we will stick with that approach here. Let us assume the solution as

$$\tilde{y} = a_1 + a_2 x + a_3 x^2 = \sum_{j=1}^n a_j \phi_j(x) \quad (15.2.11)$$

implying that

$$\phi_1(x) = 1 \quad \phi_2(x) = x \quad \phi_3(x) = x^2 \quad (15.2.12)$$

The only difference between Eqn. (15.2.1) and the above equation is the ϕ_0 term. In this modified approach we will be applying the BCs numerically and hence there is no need for the ϕ_0 term. Now we are ready to compute the terms in Eqn. (15.2.8). Using (15.2.12)

$$\frac{d\phi_1}{dx} = 0 \quad \frac{d\phi_2}{dx} = 1 \quad \frac{d\phi_3}{dx} = 2x \quad (15.2.13)$$

We will compute a typical term in the stiffness matrix

$$K_{23} = \int_{x_a}^{x_b} (1)(x)(2x)dx = \frac{2}{3}(x_b^3 - x_a^3) \quad (15.2.14)$$

and a force term (in two parts)

$$F_2^{\text{int}} = - \int_{x_a}^{x_b} \frac{2}{x^2} x dx = -2 \ln \frac{x_b}{x_a} \quad (15.2.15)$$

and

$$F_2^{\text{bnd}} = \left(-x \frac{d \tilde{y}}{dx} \right)_{x_a} x_a - \left(-x \frac{d \tilde{y}}{dx} \right)_{x_b} x_b \quad (15.2.16)$$

where F_2^{int} is the interior load term and F_2^{bnd} is the boundary load term. Similarly, the flux can be expressed as

$$\tilde{\tau} = -x \frac{d \tilde{y}}{dx} = -a_2 x - 2a_3 x^2 \quad (15.2.17)$$

Step 5: Generate the algebraic equations as

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & \frac{1}{2}(x_b^2 - x_a^2) & \frac{2}{3}(x_b^3 - x_a^3) \\ 0 & \frac{2}{3}(x_b^3 - x_a^3) & (x_b^4 - x_a^4) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 2\left(\frac{1}{x_b} - \frac{1}{x_a}\right) \\ -2 \ln \frac{x_b}{x_a} \\ -2(x_b - x_a) \end{bmatrix} + \begin{bmatrix} \tilde{\tau}|_{x_a} - \tilde{\tau}|_{x_b} \\ \tilde{\tau}|_{x_a} x_a - \tilde{\tau}|_{x_b} x_b \\ \tilde{\tau}|_{x_a} x_a^2 - \tilde{\tau}|_{x_b} x_b^2 \end{bmatrix} \quad (15.2.18)$$

We can now proceed further by substituting the problem data.

Step 6: Substituting the numerical data.

$$x_a = 1 \quad x_b = 2$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & \frac{3}{2} & \frac{14}{3} \\ 0 & \frac{14}{3} & 15 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} -1 \\ -2 \ln 2 \\ -2 \end{bmatrix} + \begin{bmatrix} \tilde{\tau}|_{x=1} - \tilde{\tau}|_{x=2} \\ \tilde{\tau}|_{x=1} - \tilde{\tau}|_{x=2} (2) \\ \tilde{\tau}|_{x=1} - \tilde{\tau}|_{x=2} (4) \end{bmatrix} \quad (15.2.19)$$

Step 7: Applying the BCs

First we will apply the natural boundary condition $\left(-x \frac{dy}{dx} \right)_{x=2} = \frac{1}{2}$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & \frac{3}{2} & \frac{14}{3} \\ 0 & \frac{14}{3} & 15 \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \end{Bmatrix} = \begin{Bmatrix} -1 \\ -2 \ln 2 \\ -2 \end{Bmatrix} + \begin{Bmatrix} \tilde{\tau}|_{x=1} - \frac{1}{2} \\ \tilde{\tau}|_{x=1} - 1 \\ \tilde{\tau}|_{x=1} - 2 \end{Bmatrix} \quad (15.2.20)$$

Since the NBC is applied to the system equations it does not guarantee that the final solution will satisfy the NBC. In the classical approach, we enforced the NBC up front on the trial solution itself so that the final solution did satisfy the NBC.

Applying the EBC $y(x=1) = 2$ is different than what we saw in the Direct Stiffness Method (Topic 2). This is because there is no equation directly associated with $y(x)$. Imposing the EBC on the trial solution itself (Eqn. (15.2.11))

$$a_1 + a_2 + a_3 = 2 \quad (15.2.21)$$

We can write a_3 in terms of the other two parameters as

$$a_3 = 2 - a_1 - a_2 \quad (15.2.22)$$

Eliminating a_3 from Eqns. (15.2.20) using the above equation, we have

$$\begin{bmatrix} 0 & 0 \\ -\frac{14}{3} & \frac{3}{2} - \frac{14}{3} \\ -15 & \frac{14}{3} - 15 \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \end{Bmatrix} = \begin{Bmatrix} \tilde{\tau}|_{x=1} - \frac{3}{2} \\ \tilde{\tau}|_{x=1} - 2 \ln 2 - \frac{31}{3} \\ \tilde{\tau}|_{x=1} - 34 \end{Bmatrix} \quad (15.2.23)$$

Eliminating the last equation (from above, Eqn. (1)-Eqn. (3), Eqn.(2)-Eqn. (3))

$$\begin{bmatrix} 15 & \frac{31}{3} \\ \frac{31}{3} & \frac{43}{6} \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \end{Bmatrix} = \begin{Bmatrix} \frac{65}{2} \\ \frac{71}{3} - 2 \ln 2 \end{Bmatrix} \quad (15.2.24)$$

The final equations are symmetric, do not contain any boundary terms and can be solved numerically.

Step 8: Solving the system equations

$$a_1 = 3.719 \quad a_2 = -2.254 \quad (15.2.25)$$

Substituting in Eqn. (15.2.22),

$$a_3 = 0.535. \quad (15.2.26)$$

Hence

$$\tilde{y} = 3.719 - 2.254x + 0.535x^2 \quad (15.2.27)$$

and

$$\tilde{\tau} = 2.254x - 1.070x^2 \quad (15.2.28)$$

While we have overcome some obstacles with this procedure, the process has still a major drawback. We again revisit the issue of imposing the boundary conditions.

The Element Concept

In the preceding section, the trial solution was assumed to be applicable for the entire problem domain. We will depart from this approach and go through the process of discretization (or, creating a mesh with elements). This will enable us to develop a very general methodology making it convenient to do a variety of things including applying the boundary conditions.

One-Element Solution

Step 4: We will once again stick with the same problem as the last section. Let us assume that the domain is discretized into a single element. Let us now assume a typical element as shown below.

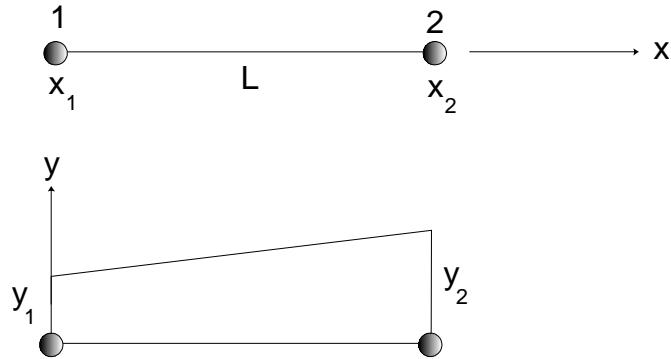


Fig. 15.2.1

The element is described by two nodes that are labeled 1 and 2 and are located at x_1 and x_2 respectively, with the length of the element as L . Let us also assume that the solution varies linearly over the element and is y_1 at node 1 and y_2 at node 2 noting that at this stage these values (called *nodal values*) are unknowns. We can describe the variation of the solution over the element as a linear **interpolation** using the two nodal values. In other words

$$\tilde{y}(x) = a_1 + a_2x = \phi_1(x)y_1 + \phi_2(x)y_2 \quad (15.2.29)$$

This is indeed the trial solution that we have used as the starting point in the preceding sections. Using the end conditions

$$y(x = x_1) = y_1 \quad y(x = x_2) = y_2 \quad (15.2.30)$$

and substituting in the first part of Eqn. (15.2.29) we obtain

$$\begin{aligned} y_1 &= a_1 + a_2x_1 \\ y_2 &= a_1 + a_2x_2 \end{aligned} \quad (15.2.31)$$

Solving for the a_i 's, we have

$$a_1 = \frac{x_2 y_1 - x_1 y_2}{x_2 - x_1} \quad a_2 = \frac{y_2 - y_1}{x_2 - x_1} \quad (15.2.32)$$

Therefore,

$$\tilde{y}(x) = \frac{x_2 y_1 - x_1 y_2}{x_2 - x_1} + \frac{y_2 - y_1}{x_2 - x_1} x \quad (15.2.33)$$

Rearranging

$$\tilde{y}(x) = \frac{x_2 - x}{x_2 - x_1} y_1 + \frac{x - x_1}{x_2 - x_1} y_2 = \frac{x_2 - x}{L} y_1 + \frac{x - x_1}{L} y_2 \quad (15.2.34)$$

The trial functions

$$\phi_1 = \frac{x_2 - x}{L} \quad \phi_2 = \frac{x - x_1}{L} \quad (15.2.35)$$

are known as the shape functions. They have special properties as shown below.

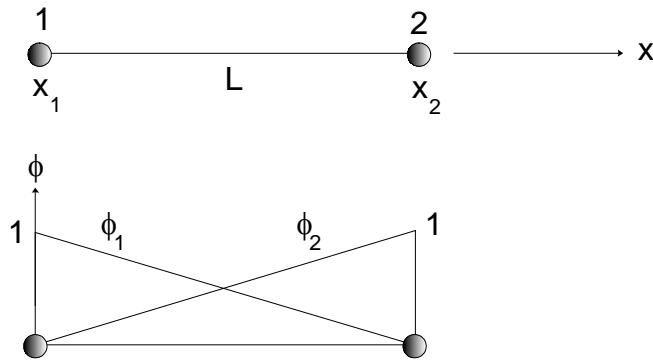


Fig. 15.2.2

Going back to Step 4 in the preceding section, we have no $\phi_0(x)$ ³ and the ϕ_i 's are given by Eqn. (15.2.35). Hence for a typical element

$$\begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \end{Bmatrix} = \begin{Bmatrix} F_1 \\ F_2 \end{Bmatrix} \quad (15.2.36)$$

Let us evaluate couple of typical terms.

$$k_{12} = \int_{x_1}^{x_2} \frac{d\phi_1}{dx} x \frac{d\phi_2}{dx} dx = \int_{x_1}^{x_2} \left(\frac{-1}{L} \right) (x) \left(\frac{1}{L} \right) dx = \frac{1}{2} \frac{x_1 + x_2}{L} \quad (15.2.37)$$

$$F_1^{\text{int}} = - \int_{x_1}^{x_2} \frac{2}{x^2} \phi_1 dx = - \frac{2}{x_1} + \frac{2}{x_2 - x_1} \ln \frac{x_2}{x_1} \quad (15.2.38)$$

³ It is not necessary to have this term since the BCs will be imposed numerically.

$$F_2^{bnd} = - \left(-x \frac{d \tilde{y}}{dx} \right)_{x_b} \quad (15.2.39)$$

Step 5: With all the other terms evaluated similarly,

$$\frac{1}{2L} \begin{bmatrix} (x_1 + x_2) & -(x_1 + x_2) \\ -(x_1 + x_2) & (x_1 + x_2) \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \end{Bmatrix} = \begin{Bmatrix} -\frac{2}{x_1} + \frac{2}{L} \ln \frac{x_2}{x_1} \\ \frac{2}{x_2} - \frac{2}{L} \ln \frac{x_2}{x_1} \end{Bmatrix} + \begin{Bmatrix} \tilde{\tau}|_{x=1} \\ -\tilde{\tau}|_{x=2} \end{Bmatrix} \quad (15.2.40)$$

These are the element equations similar to Eqn. (T2L2-4) etc. The flux expression is of the form

$$\tilde{\tau} = -x \frac{d \tilde{y}}{dx} = \frac{x}{x_2 - x_1} (y_1 - y_2) \quad (15.2.41)$$

Step 6: Substituting the numerical values

$$x_1 = 1 \quad x_2 = 2$$

we have

$$\frac{1}{2} \begin{bmatrix} 3 & -3 \\ -3 & 3 \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \end{Bmatrix} = \begin{Bmatrix} -2 + 2 \ln 2 \\ 1 - 2 \ln 2 \end{Bmatrix} + \begin{Bmatrix} \tilde{\tau}|_{x=1} \\ -\tilde{\tau}|_{x=2} \end{Bmatrix} \quad (15.2.42)$$

Step 7: Applying the boundary conditions

First we will apply the natural boundary condition $\tau|_{x=2} = \frac{1}{2}$. This results in

$$\frac{1}{2} \begin{bmatrix} 3 & -3 \\ -3 & 3 \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \end{Bmatrix} = \begin{Bmatrix} -2 + 2 \ln 2 \\ 1 - 2 \ln 2 \end{Bmatrix} + \begin{Bmatrix} \tilde{\tau}|_{x=1} \\ -\frac{1}{2} \end{Bmatrix} \quad (15.2.43)$$

Now applying the EBC $y(x = 1) = y_1 = 2$ is done in a manner described in the section containing Eqn. (T2L2-39). The equations reduce to

$$\begin{bmatrix} 1 & 0 \\ 0 & \frac{3}{2} \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \end{Bmatrix} = \begin{Bmatrix} 2 \\ \frac{7}{2} - 2 \ln 2 \end{Bmatrix} \quad (15.2.43)$$

Step 8: Solving

$$y_1 = 2 \quad y_2 = 1.409 \quad (15.2.44)$$

Hence, the approximate solution over the element is found by substituting these values in Eqns. (15.2.34) and (15.2.41) yielding

$$\tilde{y} = 2.591 - 0.591x \quad (15.2.45)$$

and

$$\tilde{\tau} = 0.591x \quad (15.2.46)$$

Comparing this solution to the exact solution shows that the results are quite in error. This is because of the linear trial solution that was assumed and because of the fact that we used only one element.

Two-Element Solution

The finite element mesh for the two-element solution is shown below.

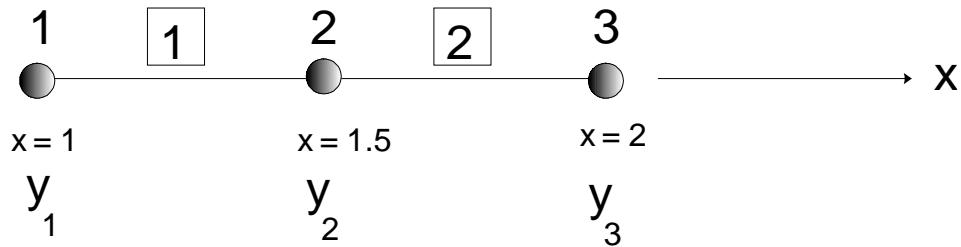


Fig. 15.2.3

This is a uniform mesh since both the elements are geometrically identical. The unknowns at the three nodes are labeled y_1, y_2, y_3 . Just as in the Direct Stiffness Method, we will generate the element equations.

Element 1: $x_1 = 1$ and $x_2 = 1.5$

$$\frac{1}{2(0.5)} \begin{bmatrix} (1+1.5) & -(1+1.5) \\ -(1+1.5) & (1+1.5) \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \end{Bmatrix} = \begin{Bmatrix} -\frac{2}{1} + \frac{2}{0.5} \ln \frac{1.5}{1} \\ \frac{2}{1.5} - \frac{2}{0.5} \ln \frac{1.5}{1} \end{Bmatrix} + \begin{Bmatrix} \left(\tilde{\tau} \Big|_{x=1} \right)_1 \\ \left(-\tilde{\tau} \Big|_{x=1.5} \right)_1 \end{Bmatrix} \quad (15.2.47)$$

Element 2: $x_1 = 1.5$ and $x_2 = 2$

$$\frac{1}{2(0.5)} \begin{bmatrix} (1.5+2) & -(1.5+2) \\ -(1.5+2) & (1.5+2) \end{bmatrix} \begin{Bmatrix} y_2 \\ y_3 \end{Bmatrix} = \begin{Bmatrix} -\frac{2}{1.5} + \frac{2}{0.5} \ln \frac{2}{1.5} \\ \frac{2}{2} - \frac{2}{0.5} \ln \frac{2}{1.5} \end{Bmatrix} + \begin{Bmatrix} \left(\tilde{\tau} \Big|_{x=1.5} \right)_2 \\ \left(-\tilde{\tau} \Big|_{x=2} \right)_2 \end{Bmatrix} \quad (15.2.48)$$

We need to expand the notation to differentiate between the two elements, i.e. $\left(\tilde{\tau} \Big|_{x=1} \right)_1$ represents the flux at $x=1$ for element 1. Assembling the two equations to create the system equations, we obtain

$$\begin{bmatrix} 2.5 & -2.5 & 0 \\ -2.5 & 6.0 & -3.5 \\ 0 & -3.5 & 3.5 \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \end{Bmatrix} = \begin{Bmatrix} -2 + 4 \ln \frac{3}{2} \\ 4 \ln \frac{8}{9} \\ 1 - 4 \ln \frac{4}{3} \end{Bmatrix} + \begin{Bmatrix} \left(\tilde{\tau} \Big|_{x=1} \right)_1 \\ 0 \\ \left(-\tilde{\tau} \Big|_{x=2} \right)_2 \end{Bmatrix} \quad (15.2.49)$$

Note that the second term in the boundary load vector is zero. This is because when we assemble the system equations we assume that

$$\left(\tilde{\tau} \Big|_{x=1.5} \right)_1 = \left(\tilde{\tau} \Big|_{x=1.5} \right)_2 \quad (15.2.50)$$

In other words the flux is assumed to be continuous across the two elements. As before, since this constraint is not enforced at the system level (but a mere substitution is made for the flux on the RHS), the final solution will **not** show this flux continuity across the elements⁴.

Now we need to impose the boundary conditions. Imposing the NBC first, we have

$$\begin{bmatrix} 2.5 & -2.5 & 0 \\ -2.5 & 6.0 & -3.5 \\ 0 & -3.5 & 3.5 \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \end{Bmatrix} = \begin{Bmatrix} -2 + 4 \ln \frac{3}{2} \\ 4 \ln \frac{8}{9} \\ 1 - 4 \ln \frac{4}{3} \end{Bmatrix} + \begin{Bmatrix} \tilde{\tau} \Big|_{x=1} \\ 0 \\ -\frac{1}{2} \end{Bmatrix} \quad (15.2.51)$$

Now imposing the EBC yields

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 6.0 & -3.5 \\ 0 & -3.5 & 6.0 \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \end{Bmatrix} = \begin{Bmatrix} 2 \\ 4 \ln \frac{8}{9} + 5 \\ \frac{1}{2} - 4 \ln \frac{4}{3} \end{Bmatrix} \quad (15.2.52)$$

Solving the equations yields

$$y_1 = 2 \quad y_2 = 1.551 \quad y_3 = 1.365 \quad (15.2.53)$$

Hence, for element 1 (using Eqns. (15.2.34) and (15.2.41))

$$\left(\tilde{y}(x) \right)_1 = 2 \left(\frac{1.5 - x}{0.5} \right) + 1.551 \left(\frac{x - 1}{0.5} \right) = -0.898x + 2.898 \quad (15.2.54)$$

$$\left(\tilde{\tau}(x) \right)_1 = 0.898x \quad (15.2.55)$$

and for element 2

⁴ This is similar to the imposition of the natural boundary condition at the system level.

$$\left(\begin{matrix} \tilde{y}(x) \\ \tilde{\tau}(x) \end{matrix} \right)_2 = 1.551 \left(\frac{2.0-x}{0.5} \right) + 1.365 \left(\frac{x-1.5}{0.5} \right) = -0.372x + 2.109 \quad (15.2.54)$$

$$\left(\begin{matrix} \tilde{y}(x) \\ \tilde{\tau}(x) \end{matrix} \right)_2 = 0.372x \quad (15.2.55)$$

Let's compare the one-element and the two-element solutions.

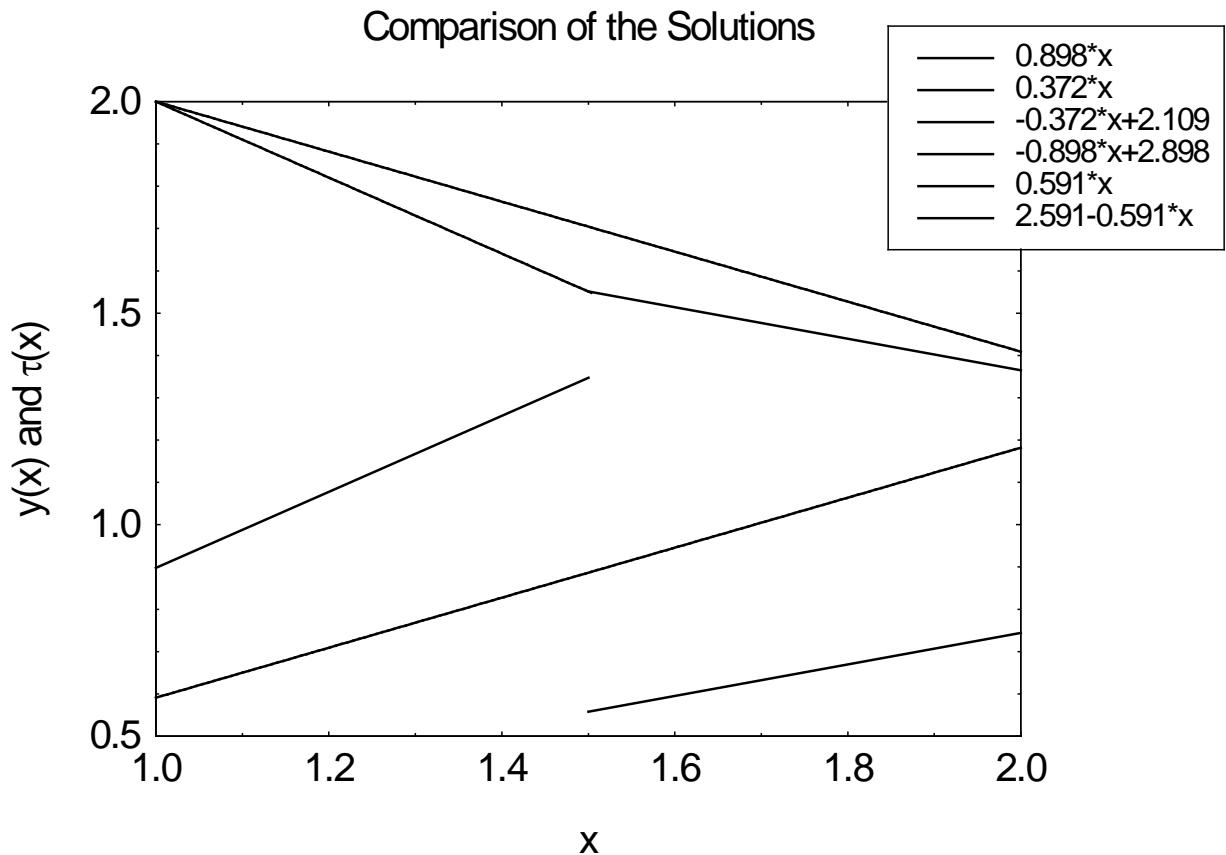


Fig. 15.2.4

- (1) Both the solutions satisfy the EBC at $x = 1$. They should, as we have imposed the EBC on the final system equations.
- (2) There is an improvement in the two-element solution with respect to $y(x)$ (An error of 4.6% and 1.3% for the one-element and two-element solutions, respectively at $x = 2$).
- (3) The error in the one-element flux is large throughout the domain. The error in the flux from the two-element solution is much less (An error of 136% and 49% for the one-element and two-element solutions, respectively at $x = 2$).
- (4) While the function is continuous throughout the domain, the flux is discontinuous for the two-element model at the element interface at $x = 1.5$.

Review of the Steps

Step 1: Assume the trial solution of the form

$$\tilde{y}(x; a) = a_0 + a_1 x + \dots + a_n x^n = \sum_{i=1}^n y_i \phi_i(x)$$

where n is equal to the number of DOF in the element, y_i are the nodal values and $\phi_i(x)$ are the shape functions. Construct the residual and substitute the trial solution in the residual equations. Note that there are as many residual equations as there are DOF in the element.

Step 2: Integrate by parts the highest derivative term. Integrating by parts will not only lower the highest derivative term but also generate a boundary (flux) term.

Step 3: Rewrite the equations so that the stiffness related terms are on the left and the load terms (interior and boundary) are on the right.

Step 4: To generate the equations completely we need to assume the exact form of the trial solution, i.e. fix the value of n . This will enable us to generate the terms in the stiffness matrix and the load vector. We can now generate the element equations in the form

$$\mathbf{k}_{n \times n} \mathbf{u}_{n \times 1} = \mathbf{f}_{n \times 1}$$

We also need to generate the expression for the flux.

Step 5: Using the problem data, we can generate the element equations for all the elements in the model. These equations are then assembled into the system or global equations of the form

$$\mathbf{K}_{N \times N} \mathbf{U}_{N \times 1} = \mathbf{F}_{N \times 1}$$

for a model with N system DOF.

Step 6: Using the problem data, we impose the NBCs first. Then we impose the EBCs resulting in equations of the form

$$\mathbf{K}_{N \times N} \mathbf{U}_{N \times 1} = \mathbf{F}_{N \times 1}$$

Step 7: Solve the system equations for the primary nodal unknowns \mathbf{U} .

Step 8: Using the primary unknowns we can compute the flux in each element.

EXERCISES

In the following problems the quadratic element is to be used. To obtain the shape functions for the quadratic element refer to Section 15.6.

Problem 15.2.1

Consider the differential equation

$$\frac{d}{dx} \left((x+1) \frac{dy(x)}{dx} \right) = 0 \quad 1 < x < 2$$

with $y(x=1) = 1 \quad \left(- (x+1) \frac{dy}{dx} \right)_{x=2} = 1$

- (a) Using the element concept, use the linear polynomial as the trial solution. Apply the steps outlined in this section to obtain an approximate solution for both the function and the flux.
- (b) Repeat the problem but now use a quadratic polynomial. Compare the two solutions.

Problem 15.2.2

Consider the differential equation

$$\frac{d}{dx} \left(x^2 \frac{dy}{dx} \right) = \frac{1}{12} (-30x^4 + 204x^3 - 351x^2 + 110x) \quad 0 < x < 4$$

with $y(0) = 1$ and $y(4) = 0$. Using a quadratic polynomial as the (element) trial solution, obtain an approximate solution for both the function and the flux. Compare this to the solution obtained earlier.

15.3 One-Dimensional Boundary Value Problem

The one-dimensional BVP is described by (with the positive x direction left to right)

$$\text{DE: } -\frac{d}{dx}\left(\alpha(x)\frac{dy(x)}{dx}\right) + \beta(x)y(x) = f(x) \quad x_a < x < x_b \quad (15.2.56)$$

$$\begin{aligned} \text{BCs: At } x = x_a \text{ either } y = y_a \text{ or } \tau = gy + c_a \\ \text{At } x = x_b \text{ either } y = y_b \text{ or } \tau = hy + c_b \end{aligned} \quad (15.2.57)$$

where $\alpha(x)$, $\beta(x)$, and $f(x)$ are known functions, y_a , y_b , c_a , c_b , g and h are constants, and $\tau = -\alpha \frac{dy}{dx}$ is the flux. The BCs are EBC, NBC or mixed. Note that the mixed BC reduces to a NBC by setting $g = 0$ and $h = 0$. We will look at specific engineering problems that are governed by this differential equation in the next lesson.

We will use the Galerkin's Method to generate the element equations. The following steps pertain to a typical element in the mesh.

Step 1: Assume the trial solution as

$$\tilde{y}(x; a) = \sum_{j=1}^n y_j \phi_j(x) \quad (15.2.58)$$

As in the previous section, we do not have the $\phi_0(x)$ term since the boundary conditions will be imposed numerically. We will drop the \sim (tilde) notation for convenience sake. Substituting in the residual equations and integrating over the domain Ω of the element, we have

$$\int_{\Omega} \left[-\frac{d}{dx} \left(\alpha(x) \frac{dy(x)}{dx} \right) + \beta(x)y(x) - f(x) \right] \phi_i(x) dx = 0 \quad i = 1, 2, \dots, n \quad (15.2.59)$$

Step 2: Integrating by parts the highest-order derivative, we have

$$\int_{\Omega} \left[\alpha(x) \frac{dy}{dx} \frac{d\phi_i}{dx} + \beta(x)y(x)\phi_i \right] dx = \int_{\Omega} f(x)\phi_i dx - [\tau\phi_i]_{\Gamma} \quad i = 1, 2, \dots, n \quad (15.2.60)$$

where Γ represents the boundary of the element.

Step 3: Using Eqn. (15.2.58) in (15.2.60)

$$\sum_{j=1}^n \left[\int_{\Omega} \frac{d\phi_i}{dx} \alpha(x) \frac{d\phi_j}{dx} dx + \int_{\Omega} \phi_i(x) \beta(x) \phi_j(x) dx \right] y_j =$$

$$\int_{\Omega} f(x)\phi_i(x) dx - [\tau\phi_i]^\Gamma \quad i = 1, 2, \dots, n \quad (15.2.61)$$

Step 4: Let us use the linear interpolation two-node element from the earlier section ($n = 2$)

$$\phi_1 = \frac{x_2 - x}{L} \quad \phi_2 = \frac{x - x_1}{L} \quad (15.2.62)$$

Substituting Eqn. (15.2.62) in Eqn. (15.2.61), we have the element equations as

$$\begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \end{Bmatrix} = \begin{Bmatrix} F_1^{\text{int}} \\ F_2^{\text{int}} \end{Bmatrix} + \begin{Bmatrix} F_1^{\text{bnd}} \\ F_2^{\text{bnd}} \end{Bmatrix} \quad (15.2.63)$$

Let us look at a typical stiffness term first.

$$k_{11} = \int_{x_1}^{x_2} \left(-\frac{1}{x_2 - x_1} \right) \alpha(x) \left(-\frac{1}{x_2 - x_1} \right) dx + \int_{x_1}^{x_2} \left(\frac{x_2 - x}{x_2 - x_1} \right) \beta(x) \left(\frac{x_2 - x}{x_2 - x_1} \right) dx \quad (15.2.64)$$

To evaluate the above equation we need to know $\alpha(x)$ and $\beta(x)$. The functions can then be substituted and the integral evaluated. Instead, we will assume that we know the numerical value of these two functions at the centroid of the element, i.e. at $x = \frac{x_1 + x_2}{2}$. For this element in which the solution is assumed to vary linearly, this is the most accurate point to evaluate the integral. Let us denote the centroidal values (constants) as $\bar{\alpha}$ and $\bar{\beta}$. Now, integrating

$$k_{11} = \frac{\bar{\alpha}}{L} + \frac{\bar{\beta}L}{3} \quad (15.2.65)$$

We can use a similar strategy with the load terms. When all the terms are evaluated we have,

$$\begin{bmatrix} \frac{\bar{\alpha}}{L} + \frac{\bar{\beta}L}{3} & -\frac{\bar{\alpha}}{L} + \frac{\bar{\beta}L}{6} \\ -\frac{\bar{\alpha}}{L} + \frac{\bar{\beta}L}{6} & \frac{\bar{\alpha}}{L} + \frac{\bar{\beta}L}{3} \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \end{Bmatrix} = \begin{Bmatrix} \frac{fL}{2} \\ \frac{fL}{2} \end{Bmatrix} - \begin{Bmatrix} [\tau\phi_1]^\Gamma \\ [\tau\phi_2]^\Gamma \end{Bmatrix} \quad (15.2.66)$$

The term that requires special treatment here is the last term. In general

$$[\tau\phi_i]^\Gamma = [\tau\phi_i]_{x_2} - [\tau\phi_i]_{x_1} \quad (15.2.67)$$

From Eqn. (15.2.57), we have

$$\tau = (gy + c) \text{ for } x = x_a \text{ and } \tau = (hy + c) \text{ for } x = x_b.$$

Substituting this in Eqn. (15.2.67) for $i = 1$

$$[\tau\phi_1]^\Gamma = [\tau\phi_1]_{x_2} - [\tau\phi_1]_{x_1} = 0 - [\tau]_{x_1} = -(gy + c)_{x_1} = -g_1 y_1 - c_1 \quad (15.2.68)$$

Similarly, for $i = 2$

$$[\tau\phi_2]^\Gamma = [\tau\phi_2]_{x_2} - [\tau\phi_2]_{x_1} = [\tau]_{x_2} - 0 = (hy + c)_{x_2} = h_2 y_2 + c_2 \quad (15.2.69)$$

Note that the unknown y appears in the two equations and must be moved to the LHS. Therefore the modified element equations are

$$\left[\begin{array}{c|c} \frac{\bar{\alpha}}{L} + \frac{\bar{\beta}L}{3} & -\frac{\bar{\alpha}}{L} + \frac{\bar{\beta}L}{6} \\ \hline -\frac{\bar{\alpha}}{L} + \frac{\bar{\beta}L}{6} & \frac{\bar{\alpha}}{L} + \frac{\bar{\beta}L}{3} \end{array} \right] - g_1 \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + h_2 \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \end{Bmatrix} = \begin{Bmatrix} \frac{\bar{f}L}{2} \\ \frac{\bar{f}L}{2} \end{Bmatrix} + \begin{Bmatrix} c_1 \\ -c_2 \end{Bmatrix} \quad (15.2.70)$$

The g_1 term exists provided $x_1 = x_a$. Similarly, h_2 term exists provided $x_2 = x_b$. The flux at the center of the element is given by

$$\tau = -\alpha \frac{dy}{dx} = -\frac{\bar{\alpha}}{L} (y_2 - y_1) \quad (15.2.71)$$

The element equations are ready and we now need to look at engineering problems that are described as 1D BVP to illustrate these steps and the rest of the solution.

Finally a warning – boundary conditions can be dangerous to your health! Applying the BCs incorrectly is one of the most common forms of errors.

15.4 Solid Mechanics

Axial Deformation of an Elastic Rod

Consider the elastic rod as shown in figure below. The axial displacement is denoted $u = u(x)$ and the axial loading on the rod is $w = w(x)$. A sample set of boundary conditions is shown where $u = 0$ (EBC) on the left end and the force F (NBC) on the right end is zero.

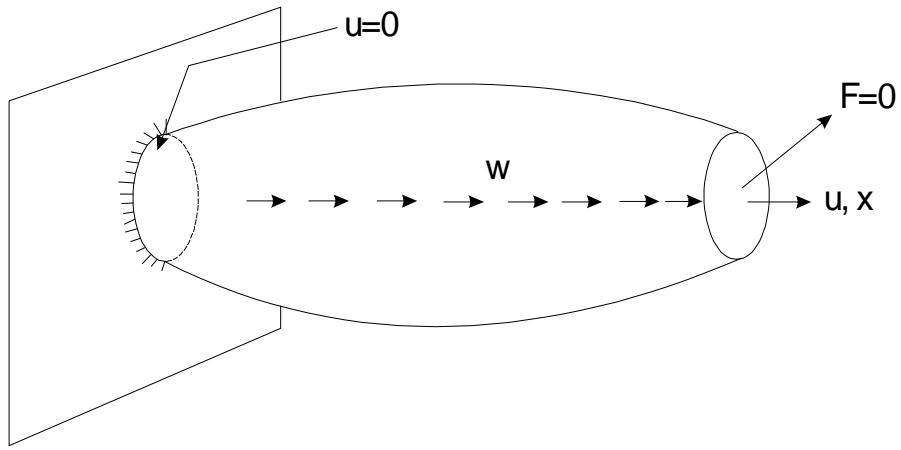


Fig. 15.4.1

The governing differential equation is given as

$$-\frac{d}{dx} \left(A(x)E(x) \frac{du(x)}{dx} \right) = w(x)A(x) \quad (15.4.1)$$

with the boundary conditions as

$$\text{At } x = x_a, u(x = x_a) = u_a \text{ or NBC} \quad (15.4.2)$$

$$\text{At } x = x_b, u(x = x_b) = u_b \text{ or NBC} \quad (15.4.3)$$

The natural BC is of the form

$$\bar{X} = n_x \sigma_x \quad (15.4.3a)$$

where n_x is the direction cosine of the outward normal, \bar{X} is the force per unit area and is positive if it acts in the positive x direction. Let us look at some possibilities with respect to the boundary conditions.

Rod has a known displacement u_a (incl. zero) at the left end

$$u = u_a$$

There is a concentrated force F_a applied at the left end in the positive x direction

$$F_a = -AE \frac{du}{dx} \Big|_{x=x_a} \quad \text{or} \quad \sigma_a = -E \frac{du}{dx} \Big|_{x=x_a}$$

There is a concentrated force F_b applied at the right end in the positive x direction

$$F_b = AE \frac{du}{dx} \Big|_{x=x_b} \quad \text{or} \quad \sigma_b = E \frac{du}{dx} \Big|_{x=x_b}$$

Using the process discussed in the previous lesson, Eqn. (15.2.70) reduces to

$$\frac{\overline{AE}}{L} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix} = \begin{Bmatrix} \frac{\overline{wAL}}{2} \\ \frac{\overline{wAL}}{2} \end{Bmatrix} + \begin{Bmatrix} F_1 \\ F_2 \end{Bmatrix} \quad (15.4.4)$$

$$\mathbf{k}_{2 \times 2} \mathbf{u}_{2 \times 1} = \mathbf{f}_{2 \times 1} \quad (15.4.5)$$

A dimensional analysis will show that - $\overline{A} \equiv L^2$, $\overline{E} \equiv \frac{F}{L^2}$, $u \equiv L$ and $\overline{wA} \equiv \frac{F}{L}$. The LHS and the

RHS have units of F . While we will look formally at systems modeled with discrete structural elements (truss and frame) in the second module, it should be noted that the stiffness matrix (in the local coordinate system aligned with the axis of the element) is the stiffness matrix for a truss element provided

- (a) the end of the element are pin connections,
- (b) the cross-sectional area and the modulus of elasticity are constant within the element, and
- (c) $f(x) = 0$ since no element loads are allowed on a truss member.

Example 15.4.1

Figure shows a bar made of a material with modulus of elasticity of 200 GPa. Compute the nodal displacements, element stresses and support reactions.

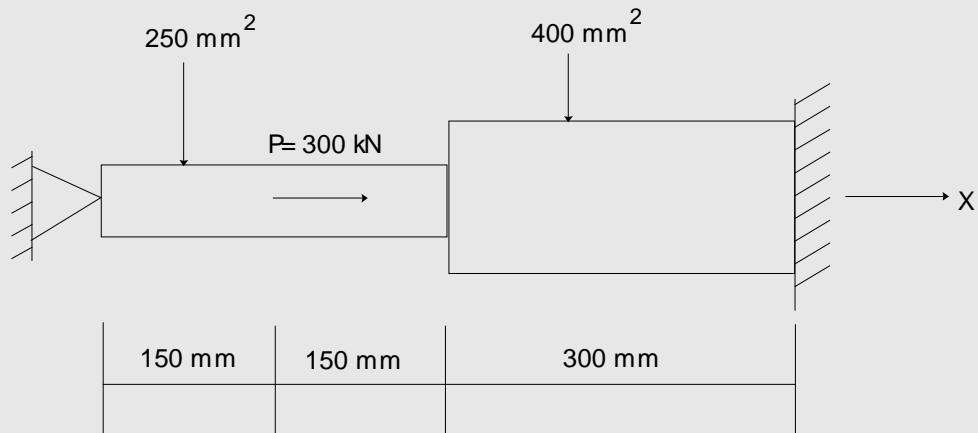


Fig. 15.4.2a

Solution: Let us use m and N as the problem units. Let us use a three-element model placing a node where the concentrated force acts⁵. The FE model is shown below. We have EBCs at the left and the right ends.



Fig. 15.4.2b

Element 1: $E = 200(10^9) \frac{N}{m^2}$, $A = 250(10^{-6}) m^2$, $L = 0.15m$, $w = 0$. Hence the element equations are

$$\begin{bmatrix} 3.333(10^8) & -3.333(10^8) \\ -3.333(10^8) & 3.333(10^8) \end{bmatrix} \begin{Bmatrix} U_1 \\ U_2 \end{Bmatrix} = \begin{Bmatrix} F_1^1 \\ -F_2^1 \end{Bmatrix}$$

Element 2: $E = 200(10^9) \frac{N}{m^2}$, $A = 250(10^{-6}) m^2$, $L = 0.15m$, $w = 0$. Hence the element equations are

$$\begin{bmatrix} 3.333(10^8) & -3.333(10^8) \\ -3.333(10^8) & 3.333(10^8) \end{bmatrix} \begin{Bmatrix} U_2 \\ U_3 \end{Bmatrix} = \begin{Bmatrix} F_1^2 \\ -F_2^2 \end{Bmatrix}$$

Element 3: $E = 200(10^9) \frac{N}{m^2}$, $A = 400(10^{-6}) m^2$, $L = 0.30m$, $w = 0$. Hence the element equations are

$$\begin{bmatrix} 2.667(10^8) & -2.667(10^8) \\ -2.667(10^8) & 2.667(10^8) \end{bmatrix} \begin{Bmatrix} U_3 \\ U_4 \end{Bmatrix} = \begin{Bmatrix} F_1^3 \\ -F_2^3 \end{Bmatrix}$$

Assembling the equations, we have

⁵ This is not necessary since we can use a Dirac Delta function $w(x)$ to define the concentrated force and then compute the equivalent forces acting at the nodes of the element.

$$10^8 \begin{bmatrix} 3.333 & -3.333 & 0 & 0 \\ -3.333 & 6.667 & -3.333 & 0 \\ 0 & -3.333 & 6 & -6.667 \\ 0 & 0 & -6.667 & 6.667 \end{bmatrix} \begin{Bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{Bmatrix} = \begin{Bmatrix} F_1^1 \\ 300(10^3) \\ 0 \\ F_2^3 \end{Bmatrix}$$

Imposing the boundary conditions, we have

$$10^8 \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 6.667 & -3.333 & 0 \\ 0 & -3.333 & 6 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 300(10^3) \\ 0 \\ 0 \end{Bmatrix}$$

Solving, the nodal displacements are

$$\{U_1, U_2, U_3, U_4\} = \{0, 6.23, 3.46, 0\}(10^{-4}) m$$

The force in each element is computed as follows. Note that flux is negative of the member force.

Element 1

$$F_1 = \frac{A_1 E_1}{L_1} (U_2 - U_1) = \frac{5(10^7)}{0.15} (6.23)(10^{-4}) = 2.077(10^5) N \text{ (Tension)}$$

This should also be equal and opposite to the support reaction at the left end, i.e. $R_{left} = 2.077(10^5) N \leftarrow$.

Element 2

$$F_2 = \frac{A_2 E_2}{L_2} (U_3 - U_2) = \frac{5(10^7)}{0.15} (3.46 - 6.23)(10^{-4}) = -9.2333(10^4) N \text{ (Compression)}$$

Element 3

$$F_3 = \frac{A_3 E_3}{L_3} (U_4 - U_3) = \frac{8(10^7)}{0.30} (0 - 3.46)(10^{-4}) = -9.2333(10^4) N \text{ (Compression)}$$

This should also be equal and opposite to the support reaction at the right end, i.e. $R_{right} = 9.2333(10^4) N \leftarrow$.

We can now examine the results. The displacements satisfy the EBC at the left and right ends. The force equilibrium is satisfied since the sum of the two support reactions is equal to the applied force. Also the forces in elements 2 and 3 are equal.

15.5 Heat Transfer

One-Dimensional Heat Conduction and Convection Problem

Consider a long, thin rod as shown in figure below. The temperature $T = T(x)$, $q = q(x)$ is the heat flux, $k = k(x)$ is the thermal conductivity, $Q = Q(x)$ is the interior volume heat source, $A = A(x)$ is the cross-sectional area, $h = h(x)$ is the convective heat transfer coefficient, $l = l(x)$ is the circumference, T_∞ is the ambient temperature. A sample set of boundary conditions is shown with an NBC on the left end and EBC on the right end. We could also have a case involving a mixed BC on the left or right ends.

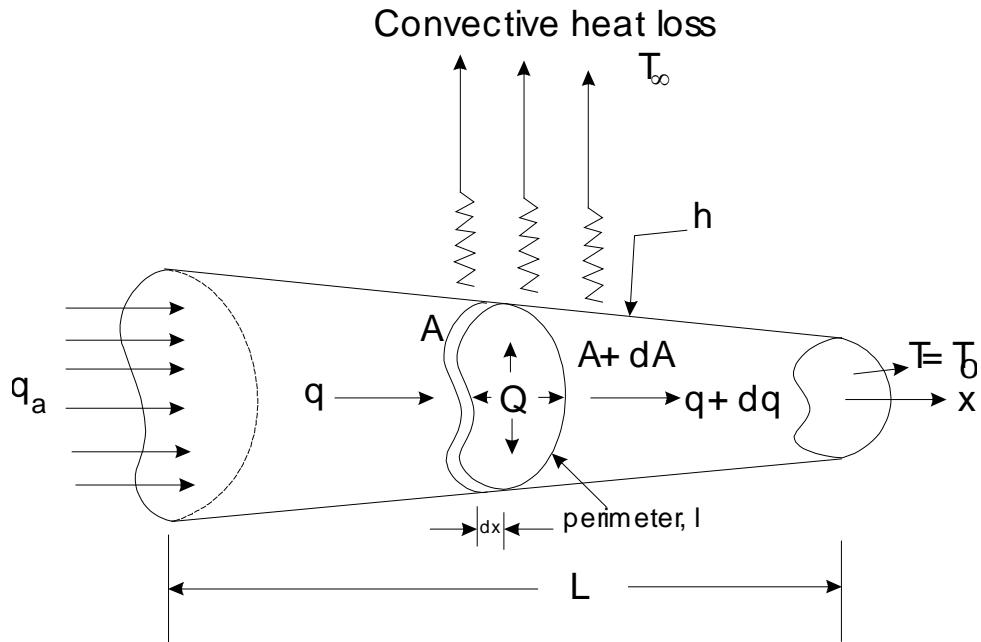


Fig. 15.5.1

The governing differential equation is given as

$$-\frac{d}{dx} \left(A(x)k(x) \frac{dT(x)}{dx} \right) + h(x)l(x)T(x) = Q(x)A(x) + h(x)l(x)T_{\infty} \quad (15.5.1a)$$

or, for a cylindrical rod with a constant cross-sectional area

$$-\frac{d}{dx} \left(k(x) \frac{dT(x)}{dx} \right) + \frac{hl}{A} T(x) = Q(x) + \frac{hl}{A} T_{\infty} \quad (15.5.1a)$$

with the boundary conditions as

$$\text{At } x = x_a, T = T_a \text{ or } -q = c_a \text{ or } -q = h_a(T - T_a^{\infty}) \quad (15.5.2a)$$

$$\text{At } x = x_b, T = T_b \text{ or } q = c_b \text{ or } q = h_b(T - T_b^{\infty}) \quad (15.5.2b)$$

Looking ahead to two and three-dimensional problems, these boundary conditions are special cases of the general form (for specified heat flow)

$$q_x n_x + q_y n_y + q_z n_z = -q_s \quad (15.5.3a)$$

if heat q_s is flowing into the surface S , and (n_x, n_y, n_z) are the direction cosines of the outward normal from the surface. Similarly, for free convection from surface S , we have

$$q_x n_x + q_y n_y + q_z n_z = h(T_s - T_{\infty}) \quad (15.5.3b)$$

Let us look at some possibilities with respect to the boundary conditions at the left end.

Left end is at a known temperature, $T = T_a$

$$T = T_a$$

Heat (q_a) is flowing into the left end

$$q = q_a$$

Left end is insulated

$$q = 0$$

Free convection is taking place at the left end (ambient temperature is T_{∞}^a and the convective coef is h_a , $T > T_{\infty}^a$)

$$-q = h_a T - h_a T_a^{\infty} \quad \text{or} \quad q = -h_a T + h_a T_a^{\infty}$$

Let us look at some possibilities with respect to the boundary conditions at the right end.

Right end is at a known temperature, $T = T_b$

$$T = T_b$$

Heat (q_b) is flowing out of the right end

$$q = q_b$$

Right end is insulated

$$q = 0$$

Free convection is taking place at the right end (ambient temperature is T_{∞}^b and the convective coef is h_b , $T > T_{\infty}^b$)

$$q = h_b T - h_b T_b^{\infty}$$

Comparing these equations to the general form of the 1D BVP⁶,

$$y(x) = T(x) \quad \alpha(x) = k(x) \quad \beta(x) = \frac{hl}{A} \quad (15.5.4a)$$

$$f(x) \equiv Q(x) + \frac{hl}{A}T_\infty \quad \tau = -k \frac{dT}{dx} = q \quad (15.5.4b)$$

$$g_1 = -h_a \quad h_2 = h_b \quad (15.5.4c)$$

Using the natural and mixed boundary conditions shown above Eqn. (15.2.70) reduces to

$$\left[\begin{array}{c|c} \frac{\bar{k}}{L} + \frac{\bar{h}L}{3A} & -\frac{\bar{k}}{L} + \frac{\bar{h}L}{6A} \\ \hline -\frac{\bar{k}}{L} + \frac{\bar{h}L}{6A} & \frac{\bar{k}}{L} + \frac{\bar{h}L}{3A} \end{array} \right] + h_1 \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + h_2 \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{Bmatrix} T_1 \\ T_2 \end{Bmatrix} = \frac{L}{2} \begin{Bmatrix} \bar{Q} + \frac{\bar{h}L}{A} T_\infty \\ \bar{Q} + \frac{\bar{h}L}{A} T_\infty \end{Bmatrix} + \begin{Bmatrix} q_1 \\ -q_2 \end{Bmatrix} + \begin{Bmatrix} h_1 T_\infty^1 \\ h_2 T_\infty^2 \end{Bmatrix} \quad (15.5.5a)$$

$$\mathbf{k}_{2 \times 2} \mathbf{u}_{2 \times 1} = \mathbf{f}_{2 \times 1} \quad (15.5.5b)$$

Note that on the LHS and the RHS some of the components will be zero depending on whether the boundary condition is NBC or mixed. A dimensional analysis will show that - $T \equiv T$, $\bar{A} \equiv L^2$, $\bar{k} \equiv \frac{E}{tL}$, $\bar{h} \equiv \frac{E}{tL^2}$, $l = L$, $\bar{Q} \equiv \frac{E}{tL^3}$, $c \equiv \frac{E}{tL^2}$, the LHS and the RHS have the units $\frac{E}{tL^2}$.

Example 15.5.1

The figure shows a composite wall made of three materials. The outer temperature is $20^\circ C$. Convection heat transfer place on the inner surface of the wall with $T_\infty = 800^\circ C$ and $h = 25 \frac{W}{m^2 \cdot ^\circ C}$.

The thermal conductivities are $k_1 = 20 \frac{W}{m \cdot ^\circ C}$, $k_2 = 30 \frac{W}{m \cdot ^\circ C}$, $k_3 = 50 \frac{W}{m \cdot ^\circ C}$. We need to determine the temperature distribution in the wall.

⁶ The sign convention is as follows – energy (or, heat flow) into the surface or boundary is positive.

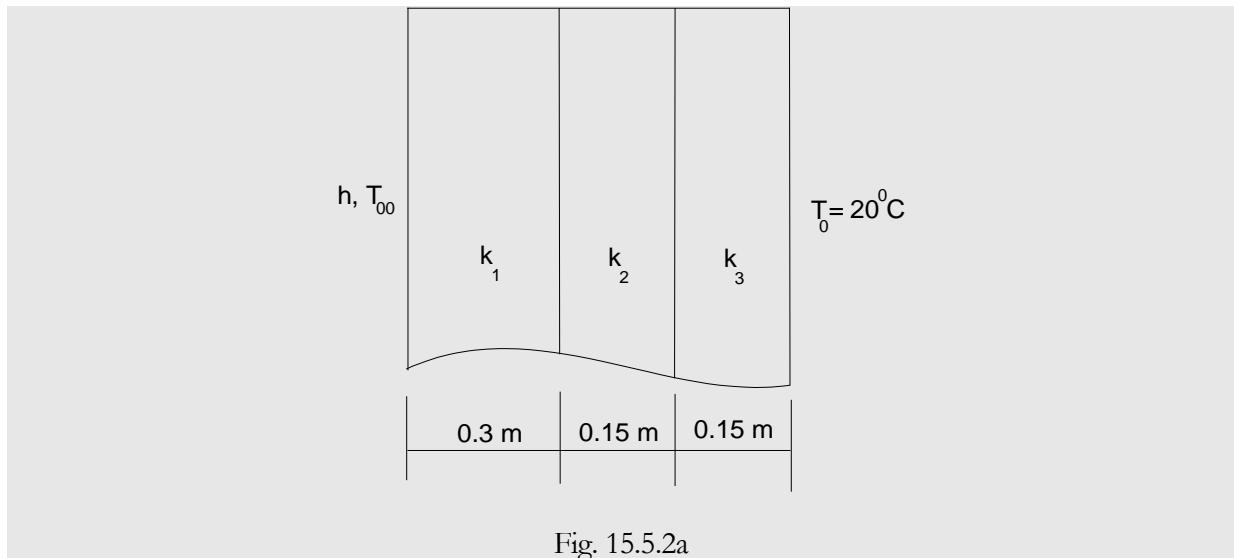


Fig. 15.5.2a

Solution: We will use a three-element model as shown. Note the following - (a) This is a problem where the convective heat exchange (gain) takes place from the left end (mixed BC). (b) There is no convective heat exchange from the top and bottom of the wall that are assumed to be very tall compared to the thickness of the wall. Hence, $h = 0 = l$. We will assume a unit area for all computations, i.e. $A = 1 \text{ m}^2$. There is no internal heat generation, i.e. $\bar{Q} = 0$. (c) The right end is at a specified temperature (EBC). For the sake of convenience, we will not include the boundary flux load terms (assuming inter-element flux continuity).

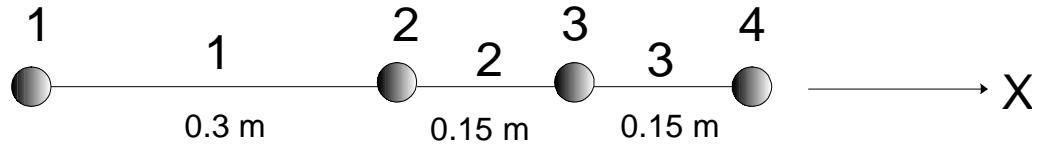


Fig. 15.5.2b

Element 1: $\bar{k} = 20 \frac{W}{m \cdot ^\circ C}$, $L = 0.3 \text{ m}$, $A = 1 \text{ m}^2$, $h = 0$, $l = 0$, $h_1 = 25 \frac{W}{m^2 \cdot ^\circ C}$, $T_1^\infty = 800^\circ\text{C}$, $h_2 = 0$, $\bar{Q} = 0$, $c_1 = 0$ and $c_2 = 0$. The element equations are as follows.

$$\begin{bmatrix} \frac{20}{0.3} + 25 & -\frac{20}{0.3} \\ -\frac{20}{0.3} & \frac{20}{0.3} \end{bmatrix} \begin{Bmatrix} T_1 \\ T_2 \end{Bmatrix} = \begin{Bmatrix} 25(800) \\ 0 \end{Bmatrix}$$

Element 2: $\bar{k} = 30 \frac{W}{m \cdot ^\circ C}$, $L = 0.15m$, $A = 1m^2$, $h = 0$, $l = 0$, $h_1 = 0$, $h_2 = 0$, $\bar{Q} = 0$, $c_1 = 0$ and $c_2 = 0$. The element equations are as follows.

$$\left[\begin{array}{c|c} \frac{30}{0.15} & -\frac{30}{0.15} \\ \hline -\frac{30}{0.15} & \frac{30}{0.15} \end{array} \right] \begin{Bmatrix} T_2 \\ T_3 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix}$$

Element 3: $\bar{k} = 50 \frac{W}{m \cdot ^\circ C}$, $L = 0.15m$, $A = 1m^2$, $h = 0$, $l = 0$, $h_1 = 0$, $h_2 = 0$, $\bar{Q} = 0$, $c_1 = 0$ and $c_2 = 0$. The element equations are as follows.

$$\left[\begin{array}{c|c} \frac{50}{0.15} & -\frac{50}{0.15} \\ \hline -\frac{50}{0.15} & \frac{50}{0.15} \end{array} \right] \begin{Bmatrix} T_3 \\ T_4 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix}$$

Assembling the equations

$$\left[\begin{array}{cccc} 91.6667 & -66.6667 & 0 & 0 \\ -66.6667 & 266.6667 & -200.0 & 0 \\ 0 & -200.0 & 533.3333 & -333.3333 \\ 0 & 0 & -333.3333 & 333.3333 \end{array} \right] \begin{Bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{Bmatrix} = \begin{Bmatrix} 20,000 \\ 0 \\ 0 \\ 0 \end{Bmatrix}$$

There is no natural boundary condition (the mixed BC was taken care of when the element equations were generated). To enforce the EBC $T_4 = 20$, we use the elimination approach. The modified equations are

$$\left[\begin{array}{cccc} 91.6667 & -66.6667 & 0 & 0 \\ -66.6667 & 266.6667 & -200.0 & 0 \\ 0 & -200.0 & 533.3333 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right] \begin{Bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{Bmatrix} = \begin{Bmatrix} 20,000 \\ 0 \\ 6666.6667 \\ 20 \end{Bmatrix}$$

Solving the equations we have (note decreasing temperature from node 1 to 4),

$$\{T_1, T_2, T_3, T_4\} = \{304.76, 119.05, 57.14, 20\}^{\circ}\text{C}$$

We can now compute the flux in each element.

Element 1:

$$\tau^1 = -\frac{k_1}{L_1}(T_2 - T_1) = -\frac{20}{0.3}(119.05 - 304.76) = 12380.95 \frac{W}{m^2}$$

Element 2:

$$\tau^2 = -\frac{k_2}{L_2}(T_3 - T_2) = -\frac{30}{0.15}(57.14 - 119.05) = 12380.95 \frac{W}{m^2}$$

Element 3:

$$\tau^3 = -\frac{k_3}{L_3}(T_4 - T_3) = -\frac{50}{0.15}(20.0 - 57.1) = 12380.95 \frac{W}{m^2}$$

We will now examine the results. The solution satisfies the EBC at the right end. The flux is constant throughout the domain as it should.

Example 15.5.2

Fig. 15.5.3 shows a circular cross-section pin fin. It has a diameter of 0.3125" and a length of 5". At the root, the temperature is $T_0 = 150^{\circ}\text{F}$. The ambient temperature is $T_\infty = 80^{\circ}\text{F}$, the convective coefficient is $h = 6 \frac{BTU}{h \cdot ft^2 \cdot ^{\circ}F}$, and the thermal conductivity is $k = 24.8 \frac{BTU}{h \cdot ft \cdot ^{\circ}F}$. Determine the temperature distribution in the fin.

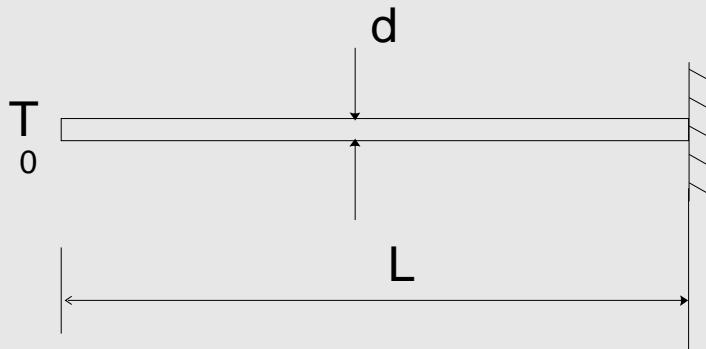


Fig. 15.5.3

Solution: We will use a two-element model to solve the problem. The FE model is shown below.



Fig. 15.5.4

Note the following - (a) In this problem the left end is tied to an EBC. (b) There is no convective heat exchange from the right end (NBC with $q = 0$). (c) There is no internal heat generation, i.e. $\bar{Q} = 0$. We will select ft as the units for length.

$$Element\ 1: \bar{k} = 24.8 \frac{BTU}{h \cdot ft \cdot {}^{\circ}F}, L = 0.208 ft, A = \pi \frac{d^2}{4} = 5.326(10^{-4}) ft^2, T_{\infty} = 80 {}^{\circ}F,$$

$h = 6 \frac{BTU}{h \cdot ft^2 \cdot {}^{\circ}F}$, $l = \pi d = 0.0818 ft$, $h_1 = 0$, $h_2 = 0$, $\bar{Q} = 0$, $c_1 = 0$ and $c_2 = 0$. The element equations are as follows.

$$\begin{bmatrix} \frac{24.8}{0.208} + \frac{6(0.0818)(0.208)}{3(5.326e-4)} & -\frac{24.8}{0.208} + \frac{6(0.0818)(0.208)}{6(5.326e-4)} \\ -\frac{24.8}{0.208} + \frac{6(0.0818)(0.208)}{6(5.326e-4)} & \frac{24.8}{0.208} + \frac{6(0.0818)(0.208)}{3(5.326e-4)} \end{bmatrix} \begin{Bmatrix} T_1 \\ T_2 \end{Bmatrix} = \frac{0.208}{2} \begin{Bmatrix} \frac{6(0.0818)}{5.326e-4}(80) \\ \frac{6(0.0818)}{5.326e-4}(80) \end{Bmatrix}$$

$$\text{or, } \begin{bmatrix} 183.12 & -87.285 \\ -87.285 & 183.12 \end{bmatrix} \begin{Bmatrix} T_1 \\ T_2 \end{Bmatrix} = \begin{Bmatrix} 7667 \\ 7667 \end{Bmatrix}$$

Element 2: Same as element 1 except we use T_2 and T_3 .

Assembling, we have

$$\begin{bmatrix} 183.12 & -87.285 & 0 \\ -87.285 & 366.25 & -87.285 \\ 0 & -87.285 & 183.12 \end{bmatrix} \begin{Bmatrix} T_1 \\ T_2 \\ T_3 \end{Bmatrix} = \begin{Bmatrix} 7667 \\ 15334 \\ 7667 \end{Bmatrix}$$

The NBC term is zero. To apply the EBC we use the elimination technique resulting in

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 366.245 & -87.285 \\ 0 & -87.285 & 183.12 \end{bmatrix} \begin{Bmatrix} T_1 \\ T_2 \\ T_3 \end{Bmatrix} = \begin{Bmatrix} 150 \\ 28427 \\ 7667 \end{Bmatrix}$$

Solving, we have

$$\{T_1, T_2, T_3\} = \{150, 98.82, 88.97\}^\circ F$$

The temperature decreases from left to right as expected. The element flux is computed as follows.

Element 1:

$$\tau^1 = -\frac{k_1}{L_1}(T_2 - T_1) = -\frac{24.8}{0.208}(98.82 - 150) = 6102 \frac{BTU}{h \cdot ft^2}$$

Element 2:

$$\tau^2 = -\frac{k_2}{L_2}(T_3 - T_2) = -\frac{24.8}{0.208}(88.97 - 98.82) = 1174 \frac{BTU}{h \cdot ft^2}$$

Let us carry out a few checks. The solution satisfies the EBC of $150^\circ F$ at $x = 0$. The flux at the right end must be zero. However, with this crude two-element model the error is large. To obtain a better accuracy we must refine the mesh, i.e. add more elements to the mesh. We will look at this aspect in the next two sections.

15.6 Higher Order Elements

So far we have seen the element concept illustrated using linear interpolation on an element described by two nodes. We also saw in the classical Galerkin's solution methodology that superior results were obtained using higher order interpolation. Can we not tie this concept to the element concept? The answer is a resounding "Yes".

Fig. 15.6.1 shows the linear element (a) on which the solution is defined by a linear polynomial, (b) is described by two nodes with nodal values y_1 and y_2 , and (c) the resulting shape functions that use the nodal values to interpolate the solution.

$$\tilde{y}(x) = a_1 + a_2 x = \phi_1(x)y_1 + \phi_2(x)y_2$$

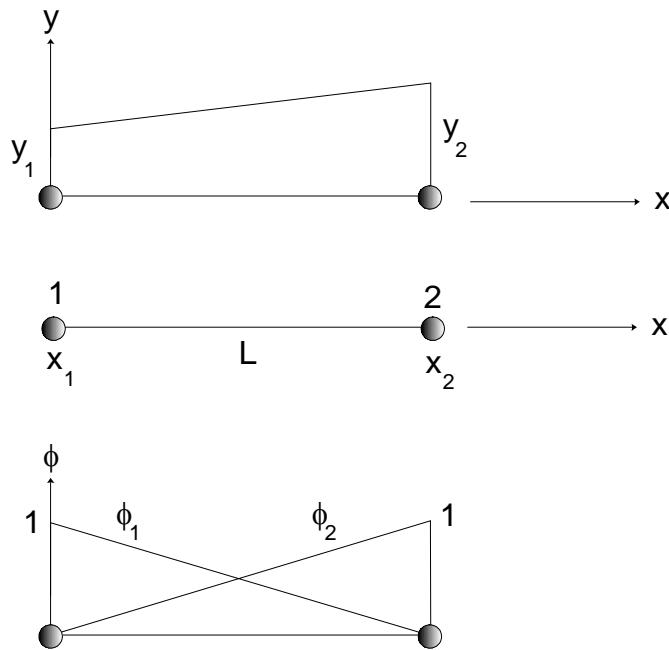


Fig. 15.6.1

Fig. 15.6.2 shows the quadratic element (a) on which the solution is defined by a quadratic polynomial, (b) is described by three nodes with nodal values y_1 , y_2 and y_3 , and (c) the resulting shape functions that use the nodal values to interpolate the solution.

$$\tilde{y}(x) = a_1 + a_2x + a_3x^2 = \phi_1(x)y_1 + \phi_2(x)y_2 + \phi_3(x)y_3 \quad (15.6.1)$$

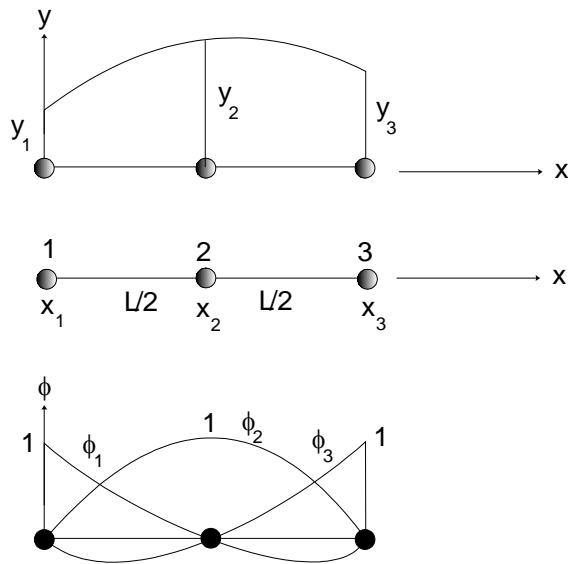


Fig. 15.6.2 Quadratic element shape functions

A few comments are in order.

- (a) For a one-dimensional element, we need to have nodes at the ends of the elements so that one element can be tied to the element next to it. So we need a minimum of two nodes.
- (b) When we have one degree of freedom per node, the number of nodes in the element is equal to the number of coefficients in the trial solution. In other words, we need a total of two nodes for the linear element and a total of three nodes for the quadratic element. Only then will we have sufficient equations to write the coefficients of the polynomials in terms of the nodal values.
- (c) The number of shape functions is also equal to the number of nodal values. We should now look at the properties of the shape functions. First, the trial solution must be complete – a quadratic polynomial must have the constant, linear **and** quadratic terms (e.g. having the constant, quadratic and cubic terms is not allowed). This is done to ensure that the element can assume all possible solution modes. Second, the shape functions satisfy the following conditions

$$\phi_i(x_j) = \delta_{ij} \quad (15.6.2)$$

where δ_{ij} is the Kronecker's Delta. In other words, the shape function will have a unit value at the node it is associate with and zeros at the other nodes (see the plots shown above). This also ensures that the shape functions are linearly independent.

Now on with the quadratic element. Using Eqn. (15.6.1) and the nodal conditions, we have

$$\begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{bmatrix} \begin{Bmatrix} a_0 \\ a_1 \\ a_2 \end{Bmatrix} = \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \end{Bmatrix} \quad (15.6.3)$$

These equations are similar to Eqn. (15.2.31). Solving for the a_i 's and collecting like terms, we have

$$\tilde{y}(x) = \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} y_1 + \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} y_2 + \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)} y_3 \quad (15.6.4)$$

Hence,

$$\phi_1(x) = \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} \quad (15.6.5a)$$

$$\phi_2(x) = \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} \quad (15.6.5b)$$

$$\phi_3(x) = \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)} \quad (15.6.5c)$$

There is an easier way to compute the shape functions that we shall see in the next module. We still have two questions to answer before we can generate the element equations for the quadratic element. First, "Where is node 2 located within the element?" Again, a detailed answer will be generated in Module 2. For the time being let us assume that it is located at the center of the element. Hence,

$$(x_2 - x_1) = (x_3 - x_2) = \frac{L}{2} \quad (x_3 - x_1) = L \quad (15.6.6)$$

Second, "How do we handle the $\alpha(x), \beta(x)$ and $f(x)$ terms?" We will develop a sophisticated technique in Module 2. For the time being let us assume that they are constants within the element. Hence,

$$\begin{aligned} & \left[\begin{array}{ccc} \frac{7\alpha}{3L} & -\frac{8\alpha}{3L} & \frac{\alpha}{3L} \\ -\frac{8\alpha}{3L} & \frac{16\alpha}{3L} & -\frac{8\alpha}{3L} \\ \frac{\alpha}{3L} & -\frac{8\alpha}{3L} & \frac{7\alpha}{3L} \end{array} \right] + \left[\begin{array}{ccc} \frac{4\beta L}{30} & \frac{2\beta L}{30} & -\frac{\beta L}{30} \\ \frac{2\beta L}{30} & \frac{16\beta L}{30} & \frac{2\beta L}{30} \\ -\frac{\beta L}{30} & \frac{2\beta L}{30} & \frac{4\beta L}{30} \end{array} \right] - g_1 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ & + h_3 \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \end{Bmatrix} = \begin{Bmatrix} \frac{fL}{6} \\ \frac{4fL}{6} \\ \frac{fL}{6} \end{Bmatrix} + \begin{Bmatrix} c_1 \\ 0 \\ -c_3 \end{Bmatrix} \end{aligned} \quad (15.6.7)$$

The flux in the element is given by

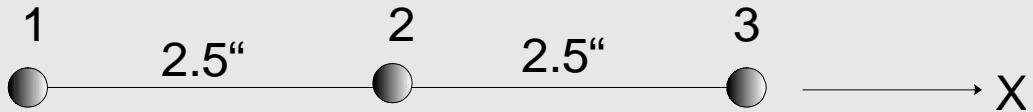
$$\begin{aligned} \tilde{\tau} &= -\alpha(x) \frac{d \tilde{y}}{dx} = -\alpha \left(\frac{2(2x - x_2 - x_3)}{L^2} y_1 + \frac{(-4)(2x - x_1 - x_3)}{L^2} y_2 + \frac{2(2x - x_1 - x_2)}{L^2} y_3 \right) \\ &= -\frac{\alpha}{L^2} [x(4y_1 - 8y_2 + 4y_3) + x_1(4y_2 - 2y_3) - 2x_2(y_1 + y_3) - 2x_3(y_1 + 2y_2)] \end{aligned} \quad (15.6.8)$$

In a similar manner we can generate higher order elements involving cubic, quartic, quintic etc. trial functions with four, five, six etc. nodes per element. The manner in which we identify these elements is usually as follows. These elements are designated as 1D-C^m interpolation order element where C^m

denotes that the problem variable and its derivatives up to order m are continuous across element boundaries. In the element formulation that we have discussed so far, only the problem variable is continuous across the element boundaries. Hence the elements are designated 1D-C⁰ linear element, or 1D-C⁰ quadratic element etc.

Example 15.6.1

Let us resolve the last problem from the previous section. This time we will use the quadratic element. Let us assume that the number of nodes is the same. The FE model is shown in Fig. 15.6.3.



1

Fig. 15.6.3

$$\text{Element 1: } \bar{k} = 24.8 \frac{\text{BTU}}{\text{h} \cdot \text{ft} \cdot {}^\circ\text{F}}, L = 0.416 \text{ ft}, A = \pi \frac{d^2}{4} = 5.326(10^{-4}) \text{ ft}^2, T_\infty = 80^\circ\text{F},$$

$h = 6 \frac{\text{BTU}}{\text{h} \cdot \text{ft}^2 \cdot {}^\circ\text{F}}, l = \pi d = 0.0818 \text{ ft}, h_1 = 0, h_2 = 0, \bar{Q} = 0, c_1 = 0 \text{ and } c_2 = 0$. The element equations are as follows ($\alpha = k, \beta = \frac{hl}{A}, f = \frac{hLT_\infty}{A}$)

$$\begin{aligned} & \left[\begin{array}{ccc} \frac{7(24.8)}{3(0.416)} & -\frac{8(24.8)}{3(0.416)} & \frac{(24.8)}{3(0.416)} \\ -\frac{8(24.8)}{3(0.416)} & \frac{16(24.8)}{3(0.416)} & -\frac{8(24.8)}{3(0.416)} \\ \frac{(24.8)}{3(0.416)} & -\frac{8(24.8)}{3(0.416)} & \frac{7(24.8)}{3(0.416)} \end{array} \right] + \\ & \left[\begin{array}{ccc} \frac{4(921.52)(0.416)}{30} & \frac{2(921.52)(0.416)}{30} & -\frac{(921.52)(0.416)}{30} \\ \frac{2(921.52)(0.416)}{30} & \frac{16(921.52)(0.416)}{30} & \frac{2(921.52)(0.416)}{30} \\ -\frac{(921.52)(0.416)}{30} & \frac{2(921.52)(0.416)}{30} & \frac{4(921.52)(0.416)}{30} \end{array} \right] \begin{Bmatrix} T_1 \\ T_2 \\ T_3 \end{Bmatrix} = \end{aligned}$$

$$\left\{ \begin{array}{l} \frac{(73721.4)(0.416)}{6} \\ \frac{4(73721.4)(0.416)}{6} \\ \frac{(73721.4)(0.416)}{6} \end{array} \right\}$$

Or,

$$\begin{bmatrix} 181.7 & -116.38 & -1.4256 \\ -116.38 & 488.33 & -116.38 \\ -1.4256 & -116.38 & 181.7 \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix} = \begin{bmatrix} 5111.3 \\ 20445 \\ 5111.3 \end{bmatrix}$$

Applying the boundary conditions (EBC for T_1), we have

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 488.33 & -116.38 \\ 0 & -116.38 & 181.7 \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix} = \begin{bmatrix} 150 \\ 37902 \\ 5325.2 \end{bmatrix}$$

Solving,

$$\{T_1, T_2, T_3\} = \{150, 99.84, 93.26\}^{\circ} F$$

The flux in the element is computed using Eqn (15.6.8) and are given by

$$\tau(x = 0.0879') = 6382 \frac{BTU}{h \cdot ft^2}$$

$$\tau(x = 0.3281') = 383.1 \frac{BTU}{h \cdot ft^2}$$

The reason for selecting the two locations will be explained in the next module. Bear in mind that the flux distribution in this element is, as Eqn. (15.6.8) shows, linear unlike the linear element in which the flux is a constant. Let's compare the two results.

	Temperature ($^{\circ}F$)		Flux $\left(\frac{BTU}{h \cdot ft^2} \right)$	
Node	Linear Elements	Quadratic Element	Linear Elements	Quadratic Element
1	150	150		
2	98.92	99.84	6102	6382
3	88.97	93.26	1174	383.1

The nodal temperatures are close but the flux values are quite different.

15.7 Mesh Refinement and Convergence

In the previous lessons we learnt two important facts. First, increasing the number of degrees of freedom increases the accuracy of the solution. In other words, if we keep on increasing the number of elements (and nodes) in the FE model, we should obtain better solutions that converge to the exact result. This is known as *h-convergence*. The *h* notation refers to the size of the elements. Second, increasing the order of the polynomial in the trial solution also increases the accuracy of the solution. In other words, if we keep on increasing the element order (and hold the number of elements constant), we should obtain better solutions that converge to the exact result. This is known as *p-convergence*. The *p* notation refers to the polynomial order. We could also combine the two and obtain what is known as *hp-convergence*.

The advantages of low-order finite elements are (a) the size of the element matrices is small, (b) the computational ease with which the elements can be generated, and (c) the size of the hand-band width of the structural stiffness matrix is small. The major disadvantage is that the convergence is slow. The comments pertaining to higher-order elements are just the opposite.

The FE mesh need not be uniform nor contain just one type of element. It may be advantageous to use the lower-order elements where the solution does not change rapidly (flux has a low value) and use the higher-order elements in regions where higher accuracy is required. By the same token, the mesh can be finer where higher accuracy is required.

The biggest disadvantage of any FE solution is the computational expense in solving the system equations. Hand calculations (with help from equation solvers in calculators) are tedious if the number of unknowns is greater than about 10. However, the finite element method is a numerical technique that is ideal for computer-based solutions. As we mentioned in the first topic, the amount of human time taken to create the data and examine the results is far greater than the time taken to solve most FE problems.

We will illustrate these ideas using the results from the computer program. If you wish you may jump to the section “Using the 1DBVP[®] Program” to familiarize yourself with the program before reading the next section.

Example 15.7.1

Let solve Example 15.5.2 using the concepts discussed earlier. We will monitor three response quantities – temperature at the right end, the flux at the left end that is the highest flux in the model and the flux at the right end that should converge to zero. The results are summarized below.

Model ID	Number of elements	Element Type	Temperature at right end	Flux at right end	Flux at left end
1	2	Linear	89	1 174	6 102
2	4	- do -	90.5	542	7 804
3	8	- do -	90.9	266	8 970
4	16	- do -	91	132	9 664
5	32	- do -	91	66	10 044
6	64	- do -	91	33	10 244
7	128	- do -	91	17	10 346
1	2	Quadratic	91.1	417	8 039
2	4	- do -	91	220	9 137
3	8	- do -	91	111	9 767
4	16	- do -	91	55	10 102
5	32	- do -	91	28	10 275
6	64	- do -	91	14	10 362

The temperature at the left end converges very rapidly to $91^{\circ}F$. The flux at the right end appears to converge linearly. The flux at the left end converges much more slowly for the linear element compared to the quadratic element.

Figs. 15.7.1 and 15.7.2 show the plots for the flux at the left and right ends for the linear and quadratic elements.

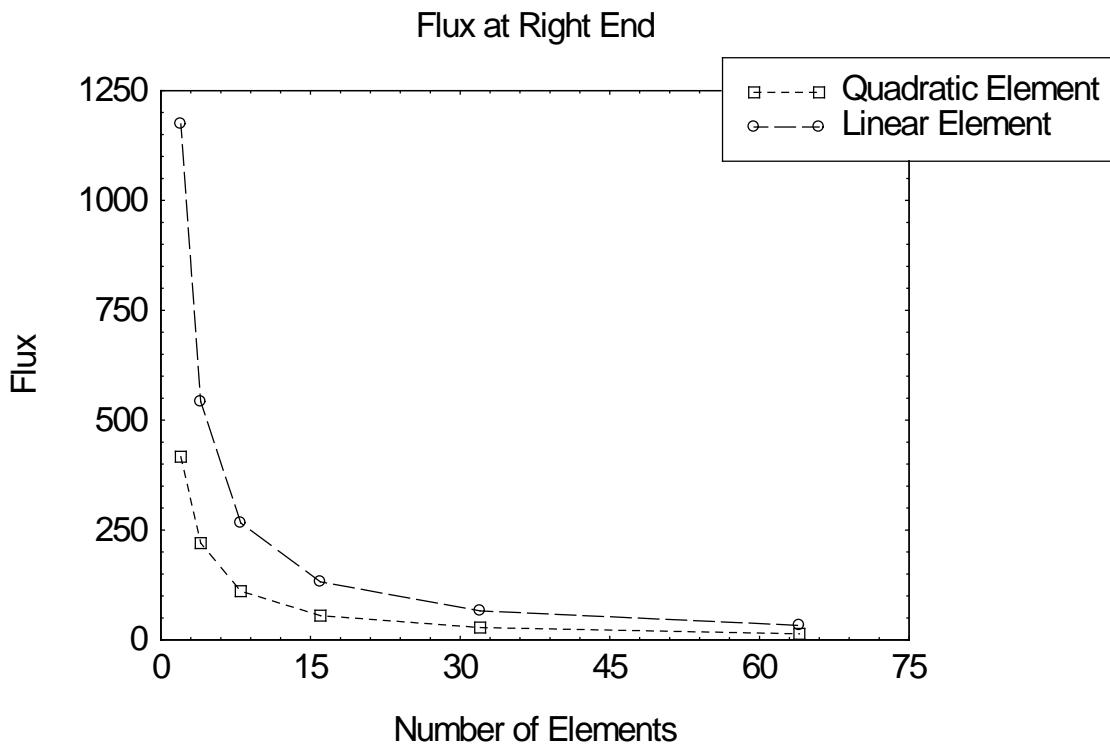


Fig. 15.7.1

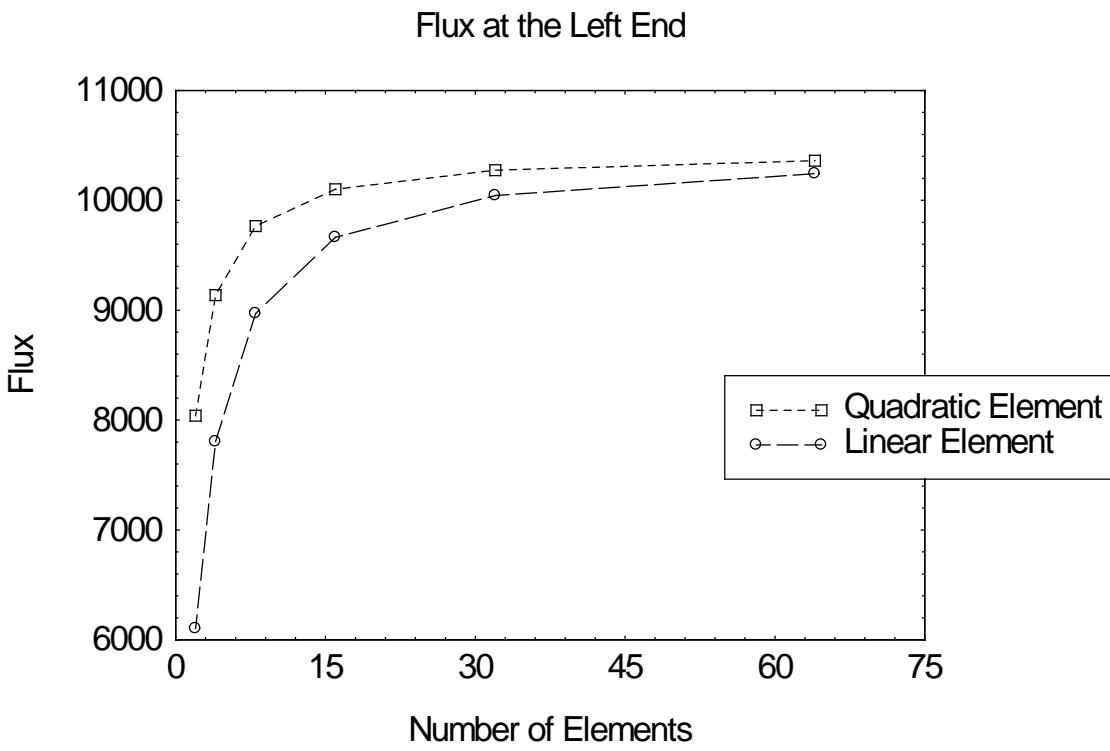


Fig. 15.7.2

We achieved two primary goals with the chapter. First, we generalized the Galerkin's Method by tying it to the element concept. This made it easier to generate the trial solutions that are valid over an arbitrary problem subdomain (the element!). The side effect is that we can handle problems with known discontinuities in the solution, e.g. the flux. Second, we looked at the manner in which the boundary conditions could be applied more easily. While the problem's essential boundary conditions were satisfied exactly, the higher-order boundary conditions were satisfied only in the limit.

The derivation of the element equations is perhaps the most important step when implementing the FE method. We saw how we could generate a family of trial solutions and the associated elements. There are two important issues that we will deal with in the next module that will make it easier to generate the element equations – a more rational way to generate the shape functions, and the isoparametric formulation.

It should be pointed out that seemingly different engineering problems are all governed by the same differential equation. However, one still needs to contend with the physics behind the parameters and the equations.

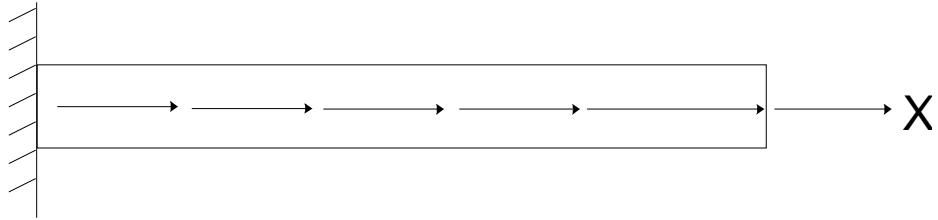
Better solutions are obtained by (a) increasing the number of elements in the mesh or, (b) by increasing the order of the element (trial solution), or (c) both. With the easy availability of FE computer programs, it is now possible to obtain accurate solutions in a relatively short period of time. The 1DBVP[©] program illustrates (in its own manner) the aspects associated with pre-processing, solution and post-processing.

EXERCISES

In all the problems below, generate all the steps by hand except solve the systems equations using a programmable calculator or a computer program. Finally check your answers using the 1DBVP[®] program. Explain the differences in answers, if any.

Problem 15.4.1

Consider the 4" bar shown below that is loaded by an axial surface traction given by the function $w(x) = x^2 \text{ lb/in.}$



The bar properties are as follows - $E = 30(10^6) \text{ psi}$ and $A = 2 \text{ in}^2$. Determine the stresses in the bar using

- (a) One linear element.
- (b) Two linear elements.
- (c) Four linear elements.

Comment on the results.

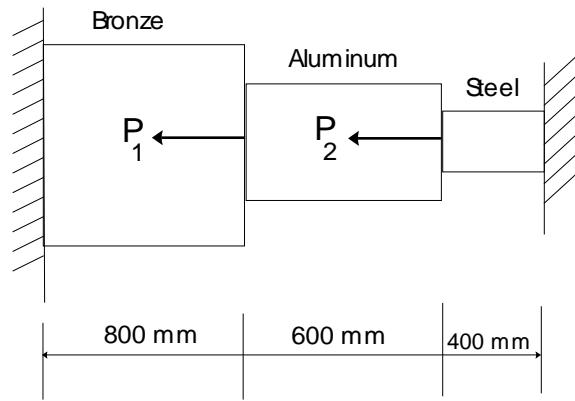
Problem 15.4.2

Resolve Problem 15.4.1 but use quadratic elements instead.

Problem 15.4.3

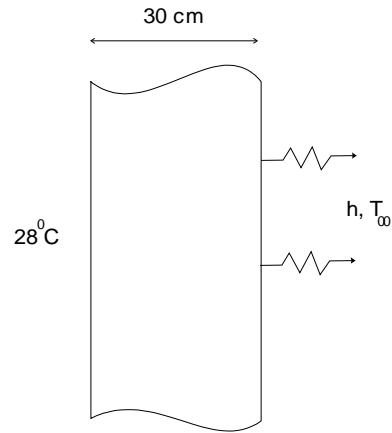
The structure shown in the figure below is subjected to an increase in temperature $\Delta T = 80^\circ C$. In addition the loads are given as follows - $P_1 = 60 \text{ kN}$, $P_2 = 75 \text{ kN}$. Determine the displacements, stresses, and support reactions using linear elements. The material properties are as follows.

Material	$A (\text{mm}^2)$	$E (\text{GPa})$	$\alpha(\text{mm}/\text{mm} \cdot {}^\circ \text{C})$
Bronze	2400	83	18.9e-6
Aluminum	1200	70	23e-6
Steel	600	200	11.7e-6



Problem 15.5.1

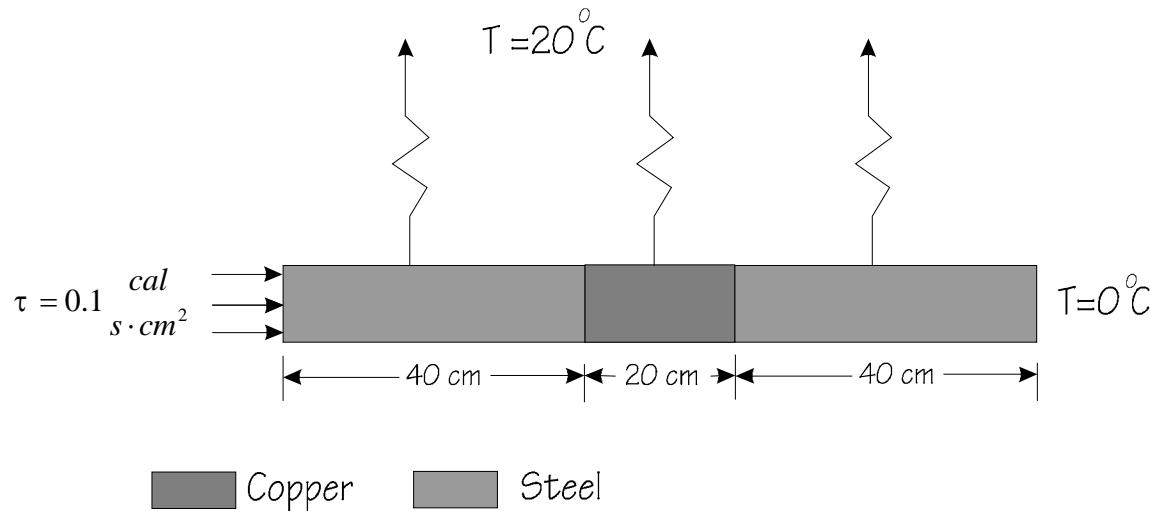
Consider a brick wall of thickness $L = 30\text{cm}$, $k = 0.7\text{W/m} \cdot ^\circ\text{C}$. The inner surface is at 28°C and the outer surface is exposed to cold air at $T_\infty = -15^\circ\text{C}$. The heat transfer coefficient associated with the outside surface is $h = 40\text{W/m}^2 \cdot ^\circ\text{C}$. Determine the steady-state temperature distribution within the wall and also the heat flux through the wall. Assume a one-dimensional heat flow. Use a two linear-element model.



Problem 15.5.2

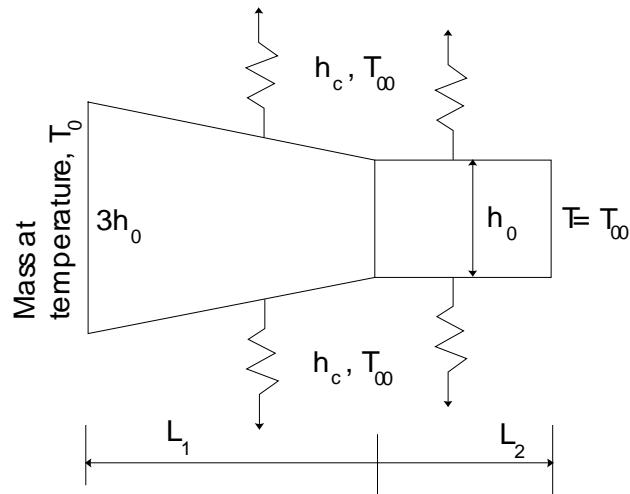
Figure shows a thin, cylindrical rod, 1m long, composed of two different materials – the two end sections each 40 cm long, are made of steel ($k = 0.12 \frac{\text{cal} - \text{cm}}{\text{s} - \text{cm}^2 - ^\circ\text{C}}$), and the center section is made of copper ($k = 0.92 \frac{\text{cal} - \text{cm}}{\text{s} - \text{cm}^2 - ^\circ\text{C}}$) is 20 cm long. The cross section is circular, with a radius of 2 cm. Heat is flowing into the left at a steady rate of $0.1\text{cal/s} - \text{cm}^2$. The temperature of the right end is maintained at a constant 0°C . The rod is in contact with air at an ambient

temperature of $20^\circ C$ so there is free convection from the lateral surface. The convective coefficient is given as $1.5 \times 10^{-4} \frac{cal}{s \cdot cm^2 \cdot {}^\circ C}$. Determine the temperature and flux distribution in the rod. Use both the linear and quadratic elements.



Problem 15.5.3

A variable area rectangular cross-section fin transmits heat away from a mass as indicated in the figure. The thickness of the fin in the direction perpendicular to the paper is ten times that shown in the plane of the paper. There is convection on the entire lateral surface.



With $h_0 = 5 \text{ cm}$, $L_1 = L_2 = 10 \text{ cm}$, $T_0 = 400^\circ C$, $T_\infty = 100^\circ C$, $h_c = 10^{-3} \text{ W/mm}^2 \cdot {}^\circ C$ and $k = 0.30 \text{ W/mm}^2 \cdot {}^\circ C$, find the temperature and flux distribution.

Eigensystems

"The person who knows "how" will always have a job. The person who knows "why" will always be his boss." Diane Ravitch

"Real learning comes about when the competitive spirit has ceased." J. Krishnamurti

"When a man sits with a pretty girl for an hour, it seems like a minute. But let him sit on a hot stove for a minute and it's longer than any hour. That's relativity." - Albert Einstein

There are several dynamic systems where energy stored in system components gives rise to the dynamic nature of the system. Examples include rotating masses, compressed springs etc. Without external excitation, the energy within the system will decay to a minimum state or will oscillate between extreme states. The rate of decay of the natural modes and the frequency of oscillation are determined by the eigenvalues of the matrix that represents the system. In this chapter we will examine eigensystems with the emphasis being on numerical techniques to compute eigenvalues and eigenvectors.

Objectives

- To understand what eigenpairs represent and their properties.
- To understand how to compute eigenpairs numerically – vector iterations methods and Jacobi Method.
- To look in some detail at two leading numerical solution techniques – Subspace Iteration Method and Lanczos Method.
- Extend our knowledge of PDEs from Chapter 17 to look at one-dimensional eigenproblem suitable for modeling engineering problems.

16.1 Eigenproblems

A square matrix $\mathbf{A}_{n \times n}$ has an eigenpair (λ, \mathbf{x}) consisting of the eigenvalue λ and the corresponding eigenvector $\mathbf{x}_{n \times 1}$ if

$$\mathbf{A}_{n \times n} \mathbf{x}_{n \times 1} = \lambda \mathbf{x}_{n \times 1} \quad (16.1.1)$$

From the above equation it should be clear that if \mathbf{x} is an eigenvector then any multiple of \mathbf{x} is also an eigenvector (though not a distinct eigenvector). Eqn. (16.1.1) is valid if and only if

$$\det(\mathbf{A} - \lambda \mathbf{I}) = 0 \quad (16.1.2)$$

Hence one way of computing the eigenpairs is to expand Eqn. (16.1.2) – an n^{th} order polynomial whose roots are the eigenvalues λ . These roots could be real or complex depending on the properties of \mathbf{A} . Finding the roots of an n^{th} order polynomial is neither easy nor efficient. Hence, in the rest of this chapter, we will see increasingly more efficient techniques to compute the eigenpairs.

The system of equations described by Eqn. (16.1.1) is called the standard eigenproblem. There is another class of problem described by

$$\mathbf{A}_{n \times n} \mathbf{X}_{n \times n} = \Lambda_{n \times n} \mathbf{B}_{n \times n} \mathbf{X}_{n \times n} \quad (16.1.3)$$

that is called the generalized eigenproblem where $\Lambda_{n \times n}$ is a diagonal matrix containing the eigenvalues and $\mathbf{X}_{n \times n}$ is a matrix where every column contains an eigenvector. Note that the generalized eigenproblem is the same as the standard eigenproblem if $\mathbf{B} = \mathbf{I}$.

Example 16.1

Compute the eigenpairs of the matrix $\begin{bmatrix} 2 & 1 \\ 2 & 6 \end{bmatrix}$.

Using Eqn. (16.1.2) we have

$$\det\left(\begin{bmatrix} 2-\lambda & 1 \\ 2 & 6-\lambda \end{bmatrix}\right) = (2-\lambda)(6-\lambda) - 2 = \lambda^2 - 8\lambda + 10 = 0$$

The quadratic polynomial has the following roots $\lambda_{1,2} = \frac{8 \pm \sqrt{64 - 40}}{2} = 1.55051, 6.44949$.

For $\lambda_1 = 1.55051$

$$\begin{bmatrix} 2 & 1 \\ 2 & 6 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = 1.55051 \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} \Rightarrow x_1 = -2.22474x_2. \text{ Or, } \mathbf{x}_{2 \times 1} = \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 1 \\ -0.449491 \end{Bmatrix}$$

For $\lambda_1 = 6.44949$

$$\begin{bmatrix} 2 & 1 \\ 2 & 6 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = 6.44949 \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} \Rightarrow x_1 = 0.224745x_2. \text{ Or, } \mathbf{x}_{2 \times 1} = \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 0.224745 \\ 1 \end{Bmatrix}$$

We have normalized both the eigenvectors by making the largest element in the eigenvector equal to 1.

To better understand some of the engineering applications, consider the spring-mass system shown in Fig. 6.1.1 that is initially at rest and where the displacement $u = u(t)$.

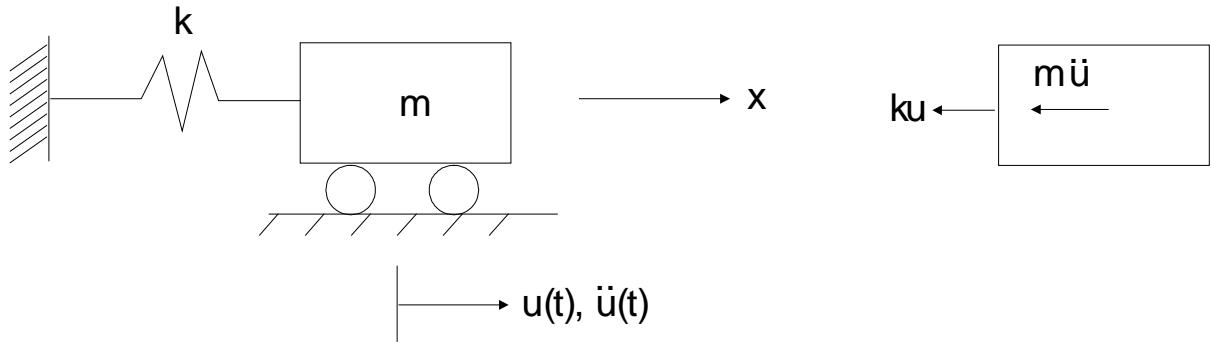


Fig. 16.1.1 System and its free-body diagram

Let us assume that the system is perturbed somehow. From the FBD, using the D'Alembert's Principle

$$-ku - m\ddot{u} = 0 \quad (16.1.4)$$

$$\text{or, } m\ddot{u} + ku = 0 \quad (16.1.5)$$

Let, $\omega^2 = \frac{k}{m}$. Substituting in the above equation we have

$$\ddot{u} + \omega^2 u = 0 \quad (16.1.6)$$

Solution to the above differential equation is of the form (sum of two harmonics)

$$u(t) = C_1 \cos \omega t + C_2 \sin \omega t \quad (16.1.7)$$

The term $\omega = \sqrt{k/m}$ is the angular frequency (expressed in rad/s), the term $f = \omega/2\pi$ is the natural frequency (expressed in Hz) and $T = 1/f$ is the natural period (expressed in s). To obtain the constants in Eqn. (16.1.7) we need the initial conditions.

At $t = t_0$, $u = u_0$ which is the initial displacement, and

$\dot{u} = \dot{u}_0$ which is the initial velocity.

Using these initial conditions, $C_1 = u_0$ and $C_2 = \frac{\dot{u}_0}{\omega}$. Hence,

$$u = u_0 \cos \omega t + \frac{\dot{u}_0}{\omega} \sin \omega t \quad (16.1.8)$$

$$\text{Or, } u = A \cos(\omega t - \alpha) \quad (16.1.9)$$

where $A = \sqrt{u_0^2 + \left(\frac{\dot{u}_0}{\omega}\right)^2}$ which is the amplitude of vibration and $\alpha = \tan^{-1} \frac{\dot{u}_0}{\omega u_0}$ is the phase angle.

Eqns. (16.1.8) and (16.1.9) represent the solution to free, undamped vibration.

Consider a harmonic forcing function

$$m\ddot{u} + ku = P \sin \Omega t \quad (16.1.10)$$

$$\text{or, } \ddot{u} + \omega^2 u = p_m \sin \Omega t \quad (16.1.11)$$

where $p_m = \frac{P}{m}$ (force per unit mass). The total solution consists of a general solution for the homogenous part and a particular solution that satisfies the whole equation. The particular solution is

$$u = C_3 \sin \Omega t \quad (16.1.12)$$

Substituting into Eqn. (16.1.11), $C_3 = \frac{p_m}{\omega^2 - \Omega^2}$. The total solution is then

$$u = C_1 \cos \omega t + C_2 \sin \omega t + C_3 \sin \Omega t \quad (16.1.13)$$

where the first two terms arise from the free vibration and the last term arises from the forced vibration. Hence,

$$u = \left[\frac{1}{1 - (\Omega/\omega)^2} \right] \frac{P}{k} \sin \Omega t \quad (16.1.14)$$

The first part of the RHS is the “magnification factor” ($1/\beta$) and the second part is the equivalent “static” load, and the solution represents the steady state forced response.

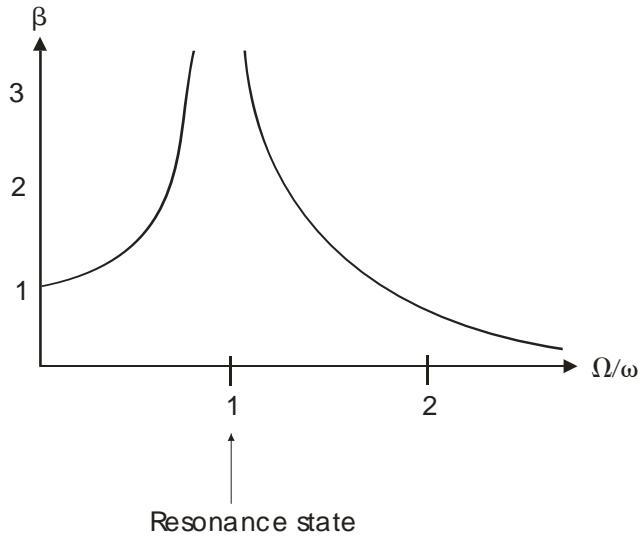


Fig. 16.1.2 Plot of β vs Ω/ω

16.2 Properties of Eigensystems

The solution to the generalized eigenproblem¹

$$\mathbf{K}_{n \times n} \mathbf{X}_{n \times n} = \boldsymbol{\Lambda}_{n \times n} \mathbf{M}_{n \times n} \mathbf{X}_{n \times n} \quad (16.2.1)$$

generates the eigenpairs – the eigenvalues and their corresponding eigenvectors. We will assume that \mathbf{K} is symmetric and positive definite whereas \mathbf{M} is symmetric. The general properties of the eigenvalue problem as described above, are as follows.

- There are n real eigenvalues and eigenvectors such that

$$0 \leq \lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n \quad (16.2.2)$$

¹ The motivation for changing the notation of the matrices from \mathbf{A}, \mathbf{B} to \mathbf{K}, \mathbf{M} is that \mathbf{K} is normally associated with stiffness and \mathbf{M} with mass both of which are associated with scientific and engineering problems.

- The eigenvector \mathbf{x}_i corresponding to the eigenvalue λ_i is such that

$$\mathbf{Kx}_i = \lambda_i \mathbf{Mx}_i \quad (16.2.3)$$

- The eigenvectors are such that

$$\mathbf{x}_i^T \mathbf{Kx}_j = 0 \quad i \neq j \quad (16.2.4)$$

$$\mathbf{x}_i^T \mathbf{Mx}_j = 0 \quad i \neq j \quad (16.2.5)$$

- The eigenvectors are generally normalized so that

$$\mathbf{x}_i^T \mathbf{Mx}_i = 1 \quad (16.2.6)$$

$$\mathbf{x}_i^T \mathbf{Kx}_i = \lambda_i \quad (16.2.7)$$

Other normalization schemes are possible such as making the largest entry in the eigenvector equal to unity. We will now look at two methods to compute the eigenpairs. These methods have limitations in terms of the size of the problem that can be handled efficiently. It should be noted that solving an eigenproblem is computationally more expensive than solving algebraic equations.

16.3 Vector Iteration Methods

These iteration methods use the properties of the Rayleigh quotient. The Rayleigh quotient $Q(\mathbf{x})$ is defined as

$$Q(\mathbf{x}) = \frac{\mathbf{x}^T \mathbf{Kx}}{\mathbf{x}^T \mathbf{Mx}} \quad (16.3.1)$$

where \mathbf{x} is an arbitrary vector. The basic property of the Rayleigh quotient is that

$$\lambda_1 \leq Q(\mathbf{x}) \leq \lambda_n \quad (16.3.2)$$

where λ_1 is the smallest eigenvalue and λ_n is the largest eigenvalue. Power iteration, inverse iteration, and subspace iteration methods use the property of Rayleigh Quotient to evaluate the eigenpairs. Power iteration can be used to find the largest eigenvalue. The inverse iteration can be used to find the lowest eigenvalue. We will see more about Subspace Iteration Method in Section 16.5.

The Inverse Iteration Method is used to evaluate the lowest eigenvalue. The process starts with an assumed (initial guess) \mathbf{x}^0 . The algorithm is as follows.

Step 1: Assume the initial guess for the eigenvector as \mathbf{x}^0 . With k as the iteration counter, set $k = 0$.

Step 2: Set $k = k + 1$.

Step 3: Solve $\mathbf{K}\hat{\mathbf{x}}^k = \mathbf{M}\mathbf{x}^{k-1}$ for $\hat{\mathbf{x}}^k$. Since the RHS will be used later, store the RHS as $\mathbf{y}^{k-1} = \mathbf{M}\mathbf{x}^{k-1}$.

Step 4: Estimate eigenvalue $\lambda^k = \frac{\hat{\mathbf{x}}^{k^T} \mathbf{y}^{k-1}}{\hat{\mathbf{x}}^{k^T} \hat{\mathbf{y}}^k}$ where $\hat{\mathbf{y}}^k = \mathbf{M}\hat{\mathbf{x}}^k$.

Step 5: Normalize eigenvector $\mathbf{x}^k = \frac{\hat{\mathbf{x}}^k}{\left(\hat{\mathbf{x}}^{k^T} \hat{\mathbf{x}}^k\right)^{1/2}}$.

Step 8: Check for convergence using $\left| \frac{\lambda^k - \lambda^{k-1}}{\lambda^k} \right| \leq \text{tolerance}$. If this condition is satisfied then \mathbf{x}^k is the eigenvector. Otherwise go to Step 2.

One should be careful in selecting the initial guess for the eigenvector \mathbf{x}^0 . Convergence will not be possible if this vector is orthogonal to the actual eigenvector. Later on we will see how it may be possible to generate the initial guess knowing the physics of the problem. Step 3 is the most expensive step in the procedure. However, if we use Cholesky Decomposition, then only the backward substitution phase needs to be used with each new RHS vector that is created every iteration.

Example 16.3.1 Inverse Iteration

Compute the eigenpairs of the following problem $\mathbf{K}_{3 \times 3} \mathbf{X}_{3 \times 3} = \Lambda_{3 \times 3} \mathbf{M}_{3 \times 3} \mathbf{X}_{3 \times 3}$ with $\mathbf{K}_{3 \times 3} = \begin{bmatrix} 3 & 2 & 1 \\ 2 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix}$

$$\text{and } \mathbf{M}_{3 \times 3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

We start with the initial guess for the eigenvector as $\mathbf{x}^0 = [1 \ 1 \ 1]^T$ and $\delta = 10^{-5}$.

Iteration 1 ($k = 1$):

$$\hat{\mathbf{x}}^k = \begin{Bmatrix} -1 \\ 2 \\ 0 \end{Bmatrix} \quad \hat{\mathbf{y}}^k = \begin{Bmatrix} -1 \\ 4 \\ 0 \end{Bmatrix} \quad \lambda^k = 0.333333 \quad \mathbf{x}^k = \begin{Bmatrix} -0.111111 \\ 0.222222 \\ 0 \end{Bmatrix}$$

Iteration 2 ($k = 2$):

$$\hat{\mathbf{x}}^k = \begin{Bmatrix} -0.555556 \\ 1 \\ -0.444444 \end{Bmatrix} \quad \hat{\mathbf{y}}^k = \begin{Bmatrix} -0.555556 \\ 2 \\ -0.444444 \end{Bmatrix} \quad \lambda^k = 0.20197 \quad \mathbf{x}^k = \begin{Bmatrix} -0.221675 \\ 0.399015 \\ -0.17734 \end{Bmatrix}$$

Iteration 3 ($k = 3$):

$$\hat{\mathbf{x}}^k = \begin{Bmatrix} -1.0197 \\ 1.99507 \\ -1.15271 \end{Bmatrix} \quad \hat{\mathbf{y}}^k = \begin{Bmatrix} -1.0197 \\ 3.99015 \\ -1.15271 \end{Bmatrix} \quad \lambda^k = 0.195814 \quad \mathbf{x}^k = \begin{Bmatrix} -0.0987208 \\ 0.193149 \\ -0.111597 \end{Bmatrix}$$

Iteration 4 ($k = 4$):

$$\hat{\mathbf{x}}^k = \begin{Bmatrix} -0.48502 \\ 0.982916 \\ -0.609494 \end{Bmatrix} \quad \hat{\mathbf{y}}^k = \begin{Bmatrix} -0.48502 \\ 1.96583 \\ -0.609494 \end{Bmatrix} \quad \lambda^k = 0.195196 \quad \mathbf{x}^k = \begin{Bmatrix} -0.19103 \\ 0.387131 \\ -0.240055 \end{Bmatrix}$$

Iteration 5 ($k = 5$):

$$\hat{\mathbf{x}}^k = \begin{Bmatrix} -0.965292 \\ 1.97961 \\ -1.25437 \end{Bmatrix} \quad \hat{\mathbf{y}}^k = \begin{Bmatrix} -0.965292 \\ 3.95922 \\ -1.25437 \end{Bmatrix} \quad \lambda^k = 0.195133 \quad \mathbf{x}^k = \begin{Bmatrix} -0.0933285 \\ 0.191397 \\ -0.121278 \end{Bmatrix}$$

Iteration 6 ($k = 6$):

$$\hat{\mathbf{x}}^k = \begin{Bmatrix} -0.476123 \\ 0.980195 \\ -0.62535 \end{Bmatrix} \quad \hat{\mathbf{y}}^k = \begin{Bmatrix} -0.476123 \\ 1.96039 \\ -1.25437 \end{Bmatrix} \quad \lambda^k = 0.195127 \quad \mathbf{x}^k = \begin{Bmatrix} -0.1875 \\ 0.386007 \\ -0.246267 \end{Bmatrix}$$

Iteration 7 ($k = 7$):

$$\hat{\mathbf{x}}^k = \begin{Bmatrix} -0.959514 \\ 1.97779 \\ -1.26455 \end{Bmatrix} \quad \hat{\mathbf{y}}^k = \begin{Bmatrix} -0.959514 \\ 3.95559 \\ -1.26455 \end{Bmatrix} \quad \lambda^k = 0.195126 \quad \mathbf{x}^k = \begin{Bmatrix} -0.0927686 \\ 0.191219 \\ -0.12226 \end{Bmatrix}$$

Hence the solution (lowest eigenpair) is $(\lambda, \mathbf{x}) = \left(0.195126, \begin{pmatrix} -0.0927686 \\ 0.191219 \\ -0.12226 \end{pmatrix} \right)$

16.4 Transformation Methods

Transformation methods are general and useful if we are interested in computing all the eigenvalues and eigenvectors of a matrix.

16.4.1 Jacobi Method

Let us again assume that \mathbf{A} is symmetric (but not necessarily positive definite) in Eqn. (16.1.1). Solving for the eigenvalues is simpler if somehow \mathbf{A} can be converted to a diagonal matrix such that the diagonal elements are in fact the eigenvalues. Clearly this process should be such that during the diagonalization process the original eigenvalues are preserved. Let $\mathbf{P}_{n \times n}$ be a nonsingular matrix. Form $\mathbf{B}_{n \times n}$ as

$$\mathbf{B} = \mathbf{P}^{-1} \mathbf{A} \mathbf{P} \quad (16.4.1)$$

Such a transformation is called a similarity transformation and it preserves the original eigenvalues but not the eigenvectors. It should be noted that $\mathbf{P}^{-1} = \mathbf{P}^T$. By performing a sequence of orthogonal similarity transformations, the symmetric matrix \mathbf{A} can be transformed into a diagonal matrix \mathbf{B} . A rotation matrix \mathbf{R} is used in this transformation. Typically rotation matrices are such that $\mathbf{R}^T \mathbf{R} = \mathbf{I}$ and $\mathbf{R}^{-1} = \mathbf{R}^T$. Assume that we need to generate \mathbf{R} so that the resulting transformation via Eqn. (16.4.1) makes $A_{ij} = A_{ji} = 0$. This is possible if

$$R_{ii} = R_{jj} = \cos(\theta) \quad (16.4.2a)$$

$$R_{ij} = -R_{ji} = -\sin(\theta) \quad (16.4.2b)$$

In other words, one would apply the rotation matrix to carry out the following transformation

$$\mathbf{B} = \mathbf{R}^T \mathbf{A} \mathbf{R} \quad (16.4.3)$$

Since \mathbf{A} is symmetric matrix, one can also infer that \mathbf{B} is symmetric. Eqns. (16.4.2) and (16.4.3) can be solved for the rotation angle yielding

$$\theta = \frac{1}{2} \tan^{-1} \left(\frac{2A_{ij}}{A_{ii} - A_{jj}} \right) \quad \text{or, } \theta = \frac{\pi}{4} \text{ if } A_{ii} = A_{jj} \quad (16.4.4)$$

The basic idea then is to apply the rotation matrix to each off-diagonal element in \mathbf{A} making it zero one at a time and continuing the process until all the off-diagonal elements are numerically small. The algorithm is as follows.

Step 1: With k as the iteration counter, set $k = 0$. Let $\mathbf{P} = \mathbf{I}$. Set values for δ and k_{\max} .

Step 2: Scan \mathbf{A} and locate the element with the largest magnitude, A_{ij} .

Step 3: Compute θ as per Eqn. (16.4.4). Set the four elements of the \mathbf{R} matrix using Eqn. (16.4.2).

Step 4: Update \mathbf{P} and \mathbf{A} as $\mathbf{P} = \mathbf{PR}$ and $\mathbf{A} = \mathbf{R}^T \mathbf{AR}$.

Step 5: Convergence check: If $|A_{ij}|_{\max} \leq \delta$, go to Step 6. Set $k = k + 1$. If $k > k_{\max}$, go to Step 6. Otherwise go to Step 2.

Step 6: The eigenvalues are the diagonal elements of \mathbf{A} and the columns of \mathbf{P} are the eigenvectors that can then be scaled appropriately.

Example 16.4.1 Jacobi Method

Compute the eigenpairs of the matrix $\begin{bmatrix} 2 & -3 \\ -3 & 10 \end{bmatrix}$.

This is a simple problem since there is only one off-diagonal element to zero out. The calculations will converge in one iteration.

$$i=1, j=2 : \theta = \frac{1}{2} \tan^{-1} \left(\frac{2(-3)}{2-10} \right) = 0.321751$$

$$\mathbf{R}_{2 \times 2} = \begin{bmatrix} 0.948683 & -0.316228 \\ 0.316228 & 0.948683 \end{bmatrix}$$

$$\mathbf{A} = \mathbf{R}^T \mathbf{AR} = \begin{bmatrix} 1 & 0 \\ 0 & 11 \end{bmatrix}$$

$$\mathbf{P} = \mathbf{PR} = \mathbf{IR} = \mathbf{R} = \begin{bmatrix} 0.948683 & -0.316228 \\ 0.316228 & 0.948683 \end{bmatrix}$$

$$\text{Hence } (\lambda_1, \mathbf{x}_1) = \left(1, \begin{Bmatrix} 0.948683 \\ 0.316228 \end{Bmatrix} \right) \text{ and } (\lambda_2, \mathbf{x}_2) = \left(11, \begin{Bmatrix} -0.316228 \\ 0.948683 \end{Bmatrix} \right)$$

16.4.2 Generalized Jacobi Method

The Generalized Jacobi Method is used to solve the generalized eigenproblem given by Eqn. (16.1.3). The basic idea is the same as that used in the Jacobi Method – successively use the rotation matrix to convert the off-diagonal elements in \mathbf{K} and \mathbf{M} to zero. When the results have converged after k iterations, we have the following.

$$\mathbf{P} = \mathbf{P}_1 \mathbf{P}_2 \dots \mathbf{P}_k \quad (16.4.5)$$

$$\hat{\mathbf{K}} = \mathbf{P}^T \mathbf{K} \mathbf{P} \quad (16.4.6)$$

$$\hat{\mathbf{M}} = \mathbf{P}^T \mathbf{M} \mathbf{P} \quad (16.4.7)$$

$$\mathbf{X} = \mathbf{P} \hat{\mathbf{M}}^{-1/2} \quad (16.4.8)$$

$$\Lambda = \hat{\mathbf{M}}^{-1} \hat{\mathbf{K}} \quad (16.4.9)$$

where

$$\hat{\mathbf{M}}^{-1} = \begin{bmatrix} \hat{M}_{11}^{-1} & & & 0 \\ & \hat{M}_{22}^{-1} & & \\ & .. & .. & \\ & & .. & \\ 0 & & & \hat{M}_{nn}^{-1} \end{bmatrix} \quad (16.4.10)$$

$$\hat{\mathbf{M}}^{-1/2} = \begin{bmatrix} \hat{M}_{11}^{-1/2} & & & 0 \\ & \hat{M}_{22}^{-1/2} & & \\ & .. & .. & \\ & & .. & \\ 0 & & & \hat{M}_{nn}^{-1/2} \end{bmatrix} \quad (16.4.11)$$

It should be noted that as $k \rightarrow \infty$, $\mathbf{K} \rightarrow \Lambda$ and $\mathbf{M} \rightarrow \mathbf{I}$.

One of more practical method is the threshold Jacobi Method in which the off-diagonal elements are tested against a cutoff value before it is determined whether they need to be zeroed out or not. Typically the coupling between rows and columns i and j is tested. In other words, if

$$\left(k_{ij}^2 / k_{ii} k_{jj} \right)^{1/2} < tol \quad (16.4.12)$$

is satisfied, then the rotation matrix is not generated and used. As a convergence check we use the following equations where s represents some tolerance value.

$$\frac{|K_{ii}^{(k+1)} - K_{ii}^{(k)}|}{K_{ii}^{(k+1)}} \leq 10^{-s} \quad i = 1, 2, \dots, n \quad (16.4.13)$$

$$\left[\frac{(K_{ij}^{(k+1)})^2}{K_{ii}^{(k+1)} K_{jj}^{(k+1)}} \right]^{1/2} \leq 10^{-s} \quad \forall i, j; i < j \quad (16.4.14)$$

The rotation matrix \mathbf{P}_k for the k^{th} iteration to zero out the off-diagonal elements K_{ij} and M_{ij} is defined as follows.

$$\mathbf{P}_k = \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & \ddots & & \\ & & & 1 & \alpha \\ i & & & & .. \\ & & & .. & & \\ & & & & .. & \\ & & & & & 1 \\ j & & \beta & & 1 & \\ & & & .. & & \\ & & & & & .. \\ & & & & & & 1 \end{bmatrix} \quad (16.4.15)$$

To get the two off-diagonal elements K_{ij} and M_{ij} as zero, we need

$$\begin{aligned} \alpha K_{ii} + (1 + \alpha\beta) K_{ij} + \beta K_{jj} &= 0 \\ \alpha M_{ii} + (1 + \alpha\beta) M_{ij} + \beta M_{jj} &= 0 \end{aligned} \quad (16.4.16)$$

Solving the equations yields the required values for α and β . Let

$$a = K_{ii}M_{ij} - M_{ii}K_{ij} \quad (16.4.17a)$$

$$b = K_{jj}M_{ij} - M_{jj}K_{ij} \quad (16.4.17b)$$

$$c = K_{ii}M_{jj} - M_{ii}K_{jj} \quad (16.4.17c)$$

Then

$$\alpha = \frac{-0.5c + \operatorname{sgn}(c)\sqrt{0.25c^2 + ab}}{a}$$

if $a \neq 0, b \neq 0$

$$\beta = -\frac{a\alpha}{b}$$
(16.4.18)

$$\alpha = \frac{K_{ij}}{K_{jj}}$$

if $a = 0$

$$\beta = 0$$
(16.4.19)

$$\alpha = 0$$

if $b = 0$

$$\beta = -\frac{K_{ij}}{K_{jj}}$$
(16.4.20)

If $a = b = 0$, then either Eqn. (16.4.19) or (16.4.20) may be used. The overall algorithm is the same as that used for the Jacobi Method. In the implementation, we will assume that both **K** and **M** are symmetric and positive definite.

Example 16.4.2 Generalized Jacobi Method

Compute the eigenpairs for the problem described in Example 16.3.1.

```
Iteration: 1
-----
Current eps: 0.0001
Coupling factors: (1,2) eptola 0.666667 and eptolb 0
Zeroing out: (1,2) alpha and beta are -0.732051 0.366025
Coupling factors: (1,3) eptola 0.394338 and eptolb 0
Zeroing out: (1,3) alpha and beta are -0.337414 0.427823
Coupling factors: (2,3) eptola 0.171279 and eptolb 0
Zeroing out: (2,3) alpha and beta are 0.307857 -0.682213
```

Eigenvalues
[1] 4.19297 [2] 10.195865 [3] 10.611169

Matrix K
Row No: 1
[1] 6.08392 [2] 10.114635 [3] 10.0352911
Row No: 2
[1] 10.114635 [2] 10.601011 [3] 15.55112e-017
Row No: 3
[1] 10.0352911 [2] 11.11022e-016 [3] 10.846282

Matrix M
Row No: 1
[1] 1.45098 [2] -3.78704e-017 [3] 15.55112e-017
Row No: 2
[1] -3.78704e-017 [2] 13.0685 [3] 10
Row No: 3
[1] 15.55112e-017 [2] 10 [3] 11.38469

```

Matrix X
Column No: 1
[1] 1 [2] 0.366025 [3] 0.427823
Column No: 2
[1] -0.501863 [2] 1.08425 [3] -0.682213
Column No: 3
[1] -0.56278 [2] 0.184355 [3] 1

Iteration: 2
-----
Current eps: 1e-008
Coupling factors: (1,2) eptola 0.00359392 and eptolb 3.22117e-034
Zeroing out: (1,2) alpha and beta are -0.0197619 0.00934471
Coupling factors: (1,3) eptola 0.000241811 and eptolb 1.53343e-033
Zeroing out: (1,3) alpha and beta are -0.00678753 0.00711377
Coupling factors: (2,3) eptola 9.59962e-007 and eptolb 2.83164e-037
Zeroing out: (2,3) alpha and beta are -0.000546427 0.00121105

Eigenvalues
[1] 4.19388 [2] 0.195126 [3] 0.610996

Matrix K
Row No: 1
[1] 6.08666 [2] -4.96129e-006 [3] 2.71098e-009
Row No: 2
[1] -4.96129e-006 [2] 0.598856 [3] 0
Row No: 3
[1] 2.71098e-009 [2] 5.53485e-017 [3] 0.846085

Matrix M
Row No: 1
[1] 1.45132 [2] -9.9047e-021 [3] -1.73472e-018
Row No: 2
[1] -9.9047e-021 [2] 3.06907 [3] -2.1684e-019
Row No: 3
[1] -1.73472e-018 [2] -2.1684e-019 [3] 1.38476

Matrix X
Column No: 1
[1] 0.991307 [2] 0.377469 [3] 0.428562
Column No: 2
[1] -0.522314 [2] 1.07724 [3] -0.68946
Column No: 3
[1] -0.569251 [2] 0.181213 [3] 0.997517

Iteration: 3
-----
Current eps: 1e-012
Coupling factors: (1,2) eptola 6.75287e-012 and eptolb 2.20249e-041
Zeroing out: (1,2) alpha and beta are 8.54884e-007 -4.04263e-007
Coupling factors: (1,3) eptola 1.42712e-018 and eptolb 1.49734e-036
Coupling factors: (2,3) eptola 1.06006e-029 and eptolb 1.10638e-038

Eigenvalues
[1] 4.19388 [2] 0.195126 [3] 0.610996

```

```

RESULTS =====
Eigenvalue: 0.195126
Eigenvector:
[1] -0.298145      [2] 0.614908      [3] -0.393555

Eigenvalue: 0.610996
Eigenvector:
[1] -0.483745      [2] 0.153993      [3] 0.847681

Eigenvalue: 4.19388
Eigenvector:
[1] 0.822861      [2] 0.313328      [3] 0.35574

```

16.5 Subspace Iteration

TBC

16.6 Lanczos Method

TBC

16.7 One-Dimensional Eigenproblem

The governing differential equation is of the form

$$-\frac{d}{dx} \left\{ \alpha(x) \frac{du(x)}{dx} \right\} + \beta(x)u(x) - \lambda\gamma(x)u(x) = 0 \quad x_a < x < x_b \quad (16.7.1)$$

with the boundary conditions as

$$\text{At } x_a: u(x_a) = 0 \quad \text{or} \quad \tau(x_a) = 0 \quad (16.7.2a)$$

$$\text{At } x_b: u(x_b) = 0 \quad \text{or} \quad \tau(x_b) = 0 \quad (16.7.2b)$$

A few points are in order when we compare this differential equation to the one-dimensional BVP. First, there is no driving force, i.e. $f(x) = 0$. Second, there is an additional term, $-\lambda\gamma(x)u(x)$ where $\gamma(x)$ describes a physical property of the system (usually mass or mass density) and the scalar λ is called the eigenvalue. Third, there are several solutions called eigensolutions to this problem. The eigensolutions consist of pairs of eigenfunction $u(x)$ and eigenvalue λ . Both of these are unknowns. Lastly, in the absence of driving forces, the condition of the system changes. This is the resonant or natural state where the internal energy oscillates back and forth between different forms e.g. kinetic and potential, without energy exchange with the surroundings.

Once again we will use the Galerkin's Approach to solve the problem.

Step 1: Residual Equations

In a typical element with the approximate solution as $\tilde{u} = \tilde{u}(x)$ (dropping the tilde notation for convenience)

$$\int_{\Omega} \left[-\frac{d}{dx} \left\{ \alpha(x) \frac{du(x)}{dx} \right\} + \beta(x)u(x) - \lambda\gamma(x)u(x) \right] \phi_i(x) dx = 0 \quad i = 1, 2, \dots, n \quad (16.7.3)$$

Step 2: Integrate by parts the highest order derivative

$$\begin{aligned} & \int_{\Omega} \frac{d\phi_i(x)}{dx} \alpha(x) \frac{d\tilde{u}}{dx} dx + \int_{\Omega} \phi_i(x) \beta(x) \tilde{u} dx - \lambda \int_{\Omega} \phi_i(x) \gamma(x) \tilde{u} dx = \\ & - \left[\left\{ -\alpha(x) \frac{d\tilde{u}}{dx} \right\} \phi_i(x) \right]_{x_1}^{x_n} \end{aligned} \quad (16.7.4)$$

where x_1 and x_n are the coordinates of the ends of the element. The last term must vanish since the boundary conditions either are essential or homogenous (meaning zero valued) natural BC's.

Step 3: Trial solution

Let the trial solution be represented as $\tilde{u}(x, a) = \sum_{j=1}^n a_j \phi_j(x)$. Hence

$$\begin{aligned} & \sum_{j=1}^n \left\{ \int_{\Omega} \frac{d\phi_i(x)}{dx} \alpha(x) \frac{d\phi_j(x)}{dx} dx + \int_{\Omega} \phi_i(x) \beta(x) \phi_j(x) dx \right\} a_j \\ & - \lambda \sum_{j=1}^n \left\{ \int_{\Omega} \phi_i(x) \gamma(x) \phi_j(x) dx \right\} a_j = 0 \quad i = 1, 2, \dots, n \end{aligned} \quad (16.7.5)$$

Writing the above equation in a compact form

$$\mathbf{k}_{n \times n} \mathbf{a}_{n \times 1} - \lambda \mathbf{m}_{n \times n} \mathbf{a}_{n \times 1} = \mathbf{0} \quad (16.7.6)$$

Step 4: Element equations for the 1D – C⁰ linear element

Considering the C⁰ linear element we have

$$\phi_1(x) = \frac{x_2 - x}{x_2 - x_1} \quad \text{and} \quad \phi_2(x) = \frac{x - x_1}{x_2 - x_1} \quad (16.7.7)$$

The terms in \mathbf{k} were evaluated earlier. We will handle the mass matrix here.

$$m_{11} = \int_{x_1}^{x_2} \phi_1(x) \gamma(x) \phi_1(x) dx = \int_{x_1}^{x_2} \frac{x_2 - x}{x_2 - x_1} \gamma(x) \frac{x_2 - x}{x_2 - x_1} dx = \frac{\bar{\gamma}L}{3} = m_{22}$$

$$m_{12} = \int_{x_1}^{x_2} \phi_1(x) \gamma(x) \phi_2(x) dx = \frac{\bar{\gamma}L}{6} = m_{21}$$

Hence

$$\mathbf{m}_{2 \times 2} = \frac{\bar{\gamma}L}{6} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

Once the element equations are assembled into the system equations, we obtain the system eigenproblem as

$$\mathbf{K}\Phi = \Lambda\mathbf{M}\Phi \quad (16.7.8)$$

that can then be solved for the eigenvalues Λ and the corresponding eigenvectors Φ .

16.8 Case Study: An Eigensolutions Toolbox

The objective is to build a library of functions that would incorporate the Inverse Iteration Method, the Jacobi Method, the Generalized Jacobi Method, the Subspace Iteration Method and the Lanczos Method.

Example Program 16.8.1

We will first show the main program and identify the functions that are in the matrix library.

main.cpp

```

1 #include <iostream>
2 #include <fstream>
3 #include "..\library\vectortemplate.h"
4 #include "..\library\matrixtemplate.h"
```

```

5   #include "..\library\matlib.h"
6   #include "..\library\fileio.h"
7
8   void ErrorHandler (int nCode)
9   {
10      switch (nCode)
11      {
12         case 1:
13             std::cout << "Invalid # of command line arguments.\n";
14             break;
15         case 2:
16             std::cout << "Cannot open input file.\n";
17             break;
18         case 3:
19             std::cout << "Cannot open output file.\n";
20             break;
21         case 4:
22             std::cout << "Error reading from input file.\n";
23             break;
24         default:
25             std::cout << "Unknown error.\n";
26             break;
27     }
28
29     exit (1);
30 }
31
32 int main (int argc, char* argv[])
33 {
34     enum Methods {INVITERATION, JACOBI, GENERALIZEDJACOBI,
35                   SUBSPACEITERATION, LANCZOS};
36
37     // set method to be used
38     Methods Method = GENERALIZEDJACOBI;
39
40     // input and output files
41     std::ifstream m_FileInput;
42     std::ofstream m_FileOutput;
43
44     if (argc <= 1)
45     {
46         // open the input file
47         OpenInputFileName ("Complete input file name: ",
48                           m_FileInput, std::ios::in);
49
50         // open the output file
51         OpenOutputFileName ("Complete output file name: ",
52                           m_FileOutput, std::ios::out);
53     }
54     // user has specified command line arguments
55     else if (argc == 3)
56     {
57         m_FileInput.open (argv[1], std::ios::in);
58         if (!m_FileInput)
59             ErrorHandler (2);
60         m_FileOutput.open (argv[2], std::ios::out);
61         if (!m_FileOutput)

```

```

62         ErrorHandler (3);
63     std::cout << "\nReading input data from " << argv[1] << '\n';
64     std::cout << "\nCreating results in    " << argv[2] << '\n';
65 }
66 else
67 {
68     ErrorHandler (1);
69 }
70
71 // define stiffness and mass matrices to be used
72 CMatrix<double> K, M;
73
74 // read stiffness matrix
75 if (K.Read (m_FileInput) != 0)
76     ErrorHandler (4);
77
78 // read mass matrix
79 if (M.Read (m_FileInput) != 0)
80     ErrorHandler (4);
81
82 if (Method == INVITERATION)
83 {
84     // read initial guess for eigenvectors
85     CMatrix<double> u;
86     if (u.Read (m_FileInput) != 0)
87         ErrorHandler (4);
88
89     // use inverse iteration method
90     double dLambda, DTOL = 1.0e-5;
91     int IterationMax = 100;
92     InverseIteration (K, M, dLambda, u, IterationMax, DTOL,
93                         true, m_FileOutput);
94     m_FileOutput << "RESULTS =====\n";
95     m_FileOutput << "Eigenvalue: " << dLambda;
96     PrintMatrixColumnWise (u, "Eigenvector: ", m_FileOutput);
97 }
98 else if (Method == GENERALIZEDJACOBI)
99 {
100     // use generalized jacobi method
101     double DTOL = 1.0e-5;
102     int IterationMax = 50;
103     const int SIZE = K.GetRows();
104     CMatrix<double> X(SIZE,SIZE);
105     CVector<double> EIGV(SIZE);
106     GeneralizedJacobi (K, M, X, EIGV, IterationMax, DTOL,
107                         true, m_FileOutput);
108     m_FileOutput << "RESULTS =====\n";
109     for (int i=1; i <= SIZE; i++)
110     {
111         m_FileOutput << "Eigenvalue: " << EIGV(i);
112         PrintMatrixColumn (X, "Eigenvector: ", i, m_FileOutput);
113         m_FileOutput << '\n';
114     }
115 }
116 else if (Method == SUBSPACEITERATION)
117 {
118     std::cout << "Subspace Iteration: Unimplemented option.\n";

```

```
119      }
120      else if (Method == LANCZOS)
121      {
122          std::cout << "Lanczos Method: Unimplemented option.\n";
123      }
124
125      // close input and output files
126      m_FileInput.close();
127      m_FileOutput.close();
128
129      return 0;
130 }
```

The program is designed to work with the generalized eigenproblem. The solution method is set in line 38 (current setting is Generalized Jacobi). As the source code indicates, this version of the example supports Inverse Iteration and Generalized Jacobi methods. These two methods are implemented in the file `matlib.h`. The exercise of adding the Subspace Iteration Method and the Lanczos Method is left to the reader. The **K** and the **M** matrices are read from the input file.

Summary

In this chapter we looked at computing eigenvalues and eigenvectors using a number of numerical techniques – Inverse Iteration, Jacobi, Generalized Jacobi, Subspace Iteration and Lanczos methods.

EXERCISES

Appetizers*Problem 16.1*

Compute all the eigenpairs of the following matrices.

(a) $\mathbf{A}_{3 \times 3} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}$

(b) $\mathbf{A}_{3 \times 3} = \begin{bmatrix} 1 & 3 & 4 \\ 3 & 1 & 2 \\ 4 & 2 & 1 \end{bmatrix}$

Main Course*Problem 16.2*

Using the Inverse Iteration Method, find the lowest eigenvalue and eigenvector for the following generalized eigenproblem.

$$\begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix}_{4 \times 4} \begin{Bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \end{Bmatrix}_{4 \times 1} = \Lambda_{4 \times 4} \begin{bmatrix} 0 & & & \\ & 2 & & \\ & & 0 & \\ & & & 1 \end{bmatrix}_{4 \times 4} \begin{Bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \end{Bmatrix}_{4 \times 1}$$

Problem 16.3

Using the Generalized Jacobi Method, find all the eigenpairs for the following generalized eigenproblem.

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 2 \end{bmatrix}_{3 \times 3} \Phi_{3 \times 3} = \Lambda_{3 \times 3} \begin{bmatrix} 0.5 & & \\ & 1 & \\ & & 0.5 \end{bmatrix}_{3 \times 3} \Phi_{3 \times 3}$$

Numerical Analysis Concepts*Problem 16.4***TBC**

Numerical Optimization

"The person who knows "how" will always have a job. The person who knows "why" will always be his boss." Diane Ravitch

'Real learning comes about when the competitive spirit has ceased.' J. Krishnamurti

Engineers and scientists are quite often interested in finding a solution to problems that are described by a central goal or objective, and whose solution is restricted by constraints. Engineering design problems are examples of this scenario. When these problems are defined in terms of a small number of variables, answers can be obtained using a paper-and-pencil approach. However, for most practical problems, it is necessary to resort to numerical techniques.

The area of numerical optimization is vast. We will deal with introductory material in this chapter with particular emphasis on nonlinear programming (NLP) problems.

Objectives

- Become familiar with the language of design optimization.
- Understand how to formulate and analytically solve simple unconstrained and constrained minimization problems.
- Learn the basics of Genetic Algorithms (GA).
- Learn how to formulate and solve nonlinear programming problems using GA.

17.1 Numerical Optimization

The simplest form of a mathematical programming problem is

$$\text{Find } \mathbf{x} \in R^n \quad (17.1-1a)$$

$$\text{To minimize } f(\mathbf{x}) \quad (17.1-1b)$$

Mathematics provides us a very powerful language to express our ideas. In the above equations, \mathbf{x} represents the vector of *design variables*. These are variables we seek in order to complete the design process. The notation $\mathbf{x} \in R^n$ indicates that the design variables are real-valued and that there are n variables - x_1, x_2, \dots, x_n . The function $f(\mathbf{x})$ is the *objective function*. This is the function that drives the design process and is either directly or indirectly a function of the n design variables. Such a problem is called an *unconstrained minimization* problem.

Consider the following problem.

$$\text{Find } x$$

$$\text{To Minimize } f(x) = (x - 10)^2 + 1$$

Since there is only one design variable, the problem is referred to as a *one-dimensional unconstrained minimization problem*.

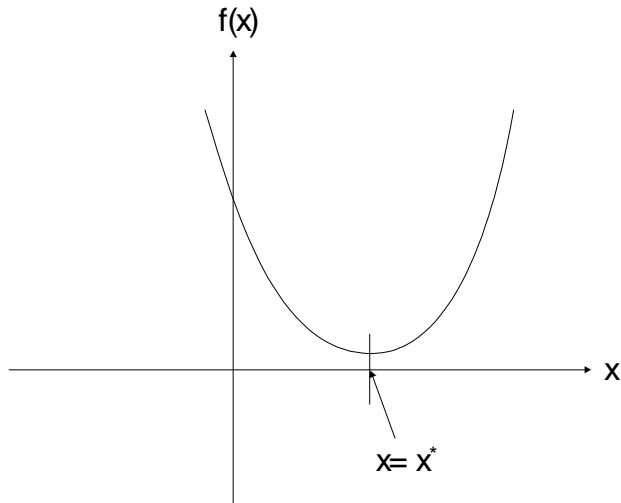


Fig. 17.1-1

From an earlier calculus course, we know for continuous differentiable functions, the necessary condition to find the minimum of a function, $f(x)$ is $\frac{df}{dx} = 0$. This condition yields the stationary

point(s). Therefore, $\frac{df}{dx} = 0 = 2(x - 10)$. Solving we have $x = 10$. The sufficient condition that this

point, $x = x^*$ corresponds to a minimum is $\frac{d^2 f(x = x^*)}{dx^2} > 0$. Checking, $\frac{d^2 f(x = x^*)}{dx^2} = 2 > 0$.

Hence, the solution to the above problem is as follows – the *optimal solution* is at $x^* = 10$ and the lowest value of the objective function is $f(x^*) = 1$. Fig. 17.1-1 shows the problem and the solution graphically.

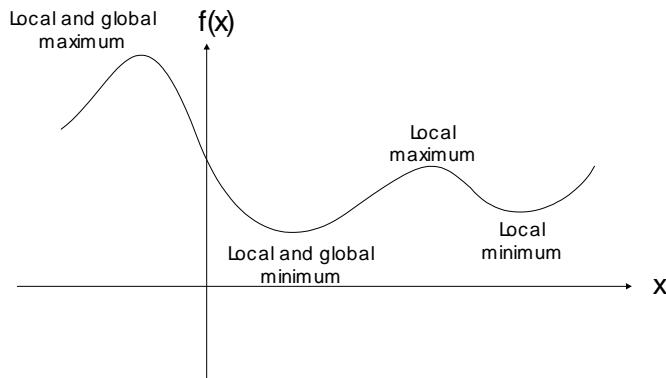


Fig. 17.1-2

There are other characteristics of even simple problems that we must be aware of. For example, in Fig. 17.1-3, the function $f(x) = x^4 - 3x^2 + x$ has multiple *local minima*. Such a function is called a *m multimodal function* (the function in Fig. 17.1-1 is a *unimodal function*). Each trough or valley captures a minimum that is local. Among all these local minima, there are one or more points that have the absolute minimum value. These points are called the *global minima*.

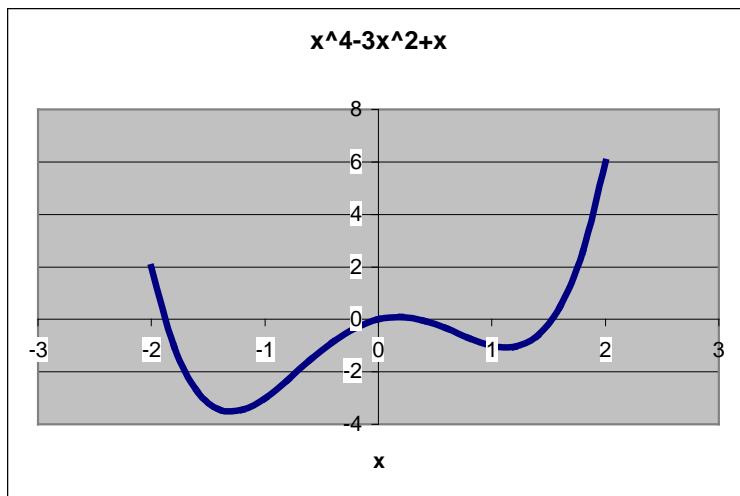


Fig. 17.1-3 A multimodal function $f(x) = x^4 - 3x^2 + x$

We can use the calculus theorems seen earlier to solve this problem.

$$\frac{df}{dx} = 0 = 4x^3 - 6x + 1 \Rightarrow x = -1.300839567, .1699384435, 1.130901123$$

Using the sufficient condition, $\frac{d^2f}{dx^2} = 12x^2 - 6$, we find that $\frac{d^2f(x = -1.300839567)}{dx^2} > 0$,

$\frac{d^2f(x = 0.1699384435)}{dx^2} < 0$, and $\frac{d^2f(x = 1.130901123)}{dx^2} > 0$. Hence the points $x^* = -1.3008$

and $x^* = 1.1309$ are local minima points. Moreover, the point $x^* = -1.3008$ is also a global minimum point. The function has the lowest value at this point. As the functions become “more nonlinear”, the task of analytically finding the minimum value becomes more difficult. One must then resort to a numerical technique.

Let us now turn our attention to two variable problems. Since there is now more than one design variable, the problem is referred to as a *multi-dimensional unconstrained minimization problem*. As an example consider the following problem.

Find $\{x_1, x_2\}$

$$\text{To Minimize}^1 \quad f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

The function is shown in Fig. 17.1.4. To find the minimum, we can use the first order condition² as

$$\frac{\partial f}{\partial x_1} = 0 = -400x_1(x_2 - x_1^2) - 2(1 - x_1)$$

$$\frac{\partial f}{\partial x_2} = 0 = 200(x_2 - x_1^2)$$

Solving, $\{x_1, x_2\} = \{1, 1\}$. The second-order condition involves computing **H** the Hessian matrix that contains the second-order (partial) derivatives of $f(\mathbf{x})$.

¹ This function is popularly known as the Rosenbrock's function.

² The syntactically correct condition is $\nabla f(\mathbf{x}) = \mathbf{0}$ where $\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & .. & \frac{\partial f}{\partial x_n} \end{bmatrix}$ is called the gradient vector.

$$\mathbf{H}_{2 \times 2} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix} = \begin{bmatrix} 1200x_1^2 - 400x_2 + 2 & -400x_1 \\ -400x_1 & 200 \end{bmatrix}$$

At the point (1,1)

$$\mathbf{H} = \begin{bmatrix} 802 & -400 \\ -400 & 200 \end{bmatrix}$$

This matrix is positive definite (all the eigenvalues are positive) implying that the point (1,1) is a minimum point.

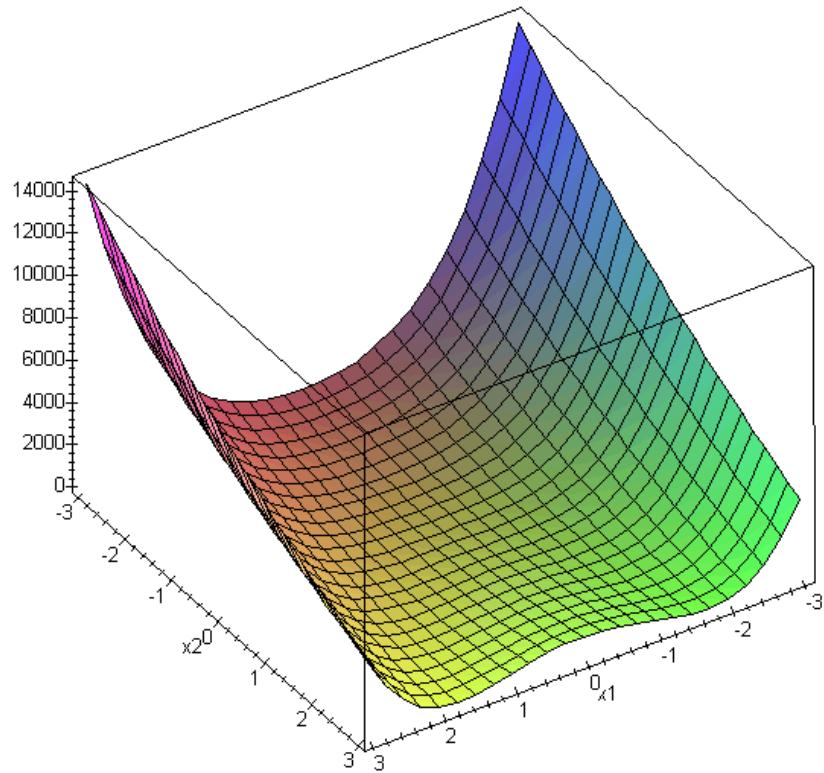


Fig. 17.1-4 Two-variable unconstrained minimization problem

The graphical solution while providing a visual look of the design space is difficult to use in locating the precise minimum. The purpose of this example is to show that the complexity of a problem increases exponentially with increasing dimension of the design space and that obtaining an analytical solution is cumbersome if not impractical.

It is difficult to formulate most engineering problems as unconstrained minimization problems. Typical engineering design problems posed in the mathematical programming format are usually of the following form.

$$\text{Find} \quad \mathbf{x} \in R^n \quad (17.1-2a)$$

$$\text{To minimize} \quad f(\mathbf{x}) \quad (17.1-2b)$$

$$\text{Subject to} \quad g_i(\mathbf{x}) \leq 0, \quad i = 1, 2, \dots, l \quad (17.1-2c)$$

$$h_j(\mathbf{x}) = 0, \quad j = 1, 2, \dots, m \quad (17.1-2d)$$

$$x_k^L \leq x_k \leq x_k^U, \quad k = 1, 2, \dots, n \quad (17.1-2e)$$

Performance requirements, manufacturing constraints or even the permissible range of values for the design variables can be specified only through *constraints*. The constraints $g_i(\mathbf{x})$ are *inequality constraints* while $h_j(\mathbf{x})$ are *equality constraints*. The constraint functions, similar to the objective function, can be linear or nonlinear, continuous, discontinuous or piecewise continuous, differentiable or non-differentiable. Eqns. (17.1-2e) establish the lower and upper bounds on the permissible values of the n design variables. These constraints are usually referred to as *bound constraints* or *side constraints*. A problem posed in the above form is called a *constrained minimization problem*. An example of the design space for a two-variable constrained problem is shown in Fig. 17.1-5.

The design variables are (x_1, x_2) . There are two inequality constraints g_1 and g_2 . The side constraints are $x_1 \geq 0$ and $x_2 \geq 0$. The hatch marks on the lines and curves representing these four constraints indicate the constraint boundary marking the barrier between the *feasible* domain and the *infeasible* domain. A feasible domain contains all the design points that satisfy all the constraints. In Fig. 17.1-5, the feasible domain is bounded by these four lines. The vertices of the feasible domain are labeled A-B-C-D. The rest of the design space is infeasible. The objective function, f is represented in the figure as isocost contours or curves that have a constant f value. As this example indicates, by sliding the isocost contours in the direction of decreasing objective function value, we can locate the optimal solution. The constraint g_1 controls the design and the optimal solution is indicated with an \mathbf{x} . The objective function has a value c_2 at this point.

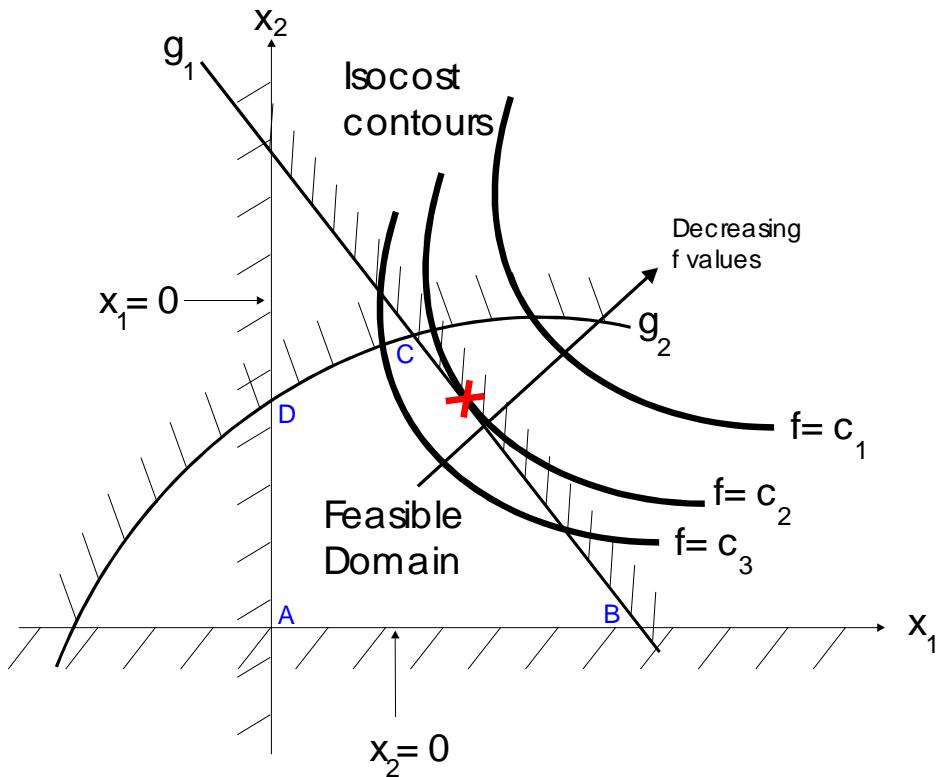


Fig. 17.1-5 Constrained design problem

Some of the different techniques to solve simple constrained minimization problems are – exhaustive search, graphical, trial and error and ‘constraint controlled’ optimal design. Exhaustive search is too expensive for solving practical problems. The graphical technique can work at most for two-variable problems. The trial and error approach is tedious and unlikely to find a local optimum. The ‘constraint controlled’ approach was ad hoc. We will formalize the approach in Section 17.3. Before we look at formalizing the approach, we will look at different types of constrained minimization problems.

17.2 Types of Mathematical Programming Problems

It would be easy to state that there is one standard type of mathematical programming (MP) problem. It would perhaps be easy then to create one or more solution techniques to solve that problem. The reality is that there are numerous types of MP problems. We will categorize the problems now.

Types of design variables: The design variables commonly encountered in engineering applications can be of different types.

Continuous design variables are those that vary continuously. For example, the height and width of a concrete beam can be taken as continuous design variables since they can, in theory, be cast in any size. Similarly, plate girders can be manufactured to any practical dimensions.

Discrete design variables are those that are available in discrete or predefined values. For example, steel I-beams are usually manufactured in predefined size and dimensions. The AISC sections are examples of such beams.

Integer design variables assume only integer values. An example of an integer design variable is the number of panel points in a roof truss. This number is a positive integer (greater than zero).

Zero-one design variables, as the name suggests, have either a zero or one value. In structural design, one could interpret the zero-one values as being equivalent to present-absent state. For example, we could designate the presence or absence of a member in a truss as being a zero-one design variable. If the design variable value is one, then the member is assumed to be a part of the truss.

Note that the properties of the design space or domain are dictated by the design variable type. With continuous design variables, the design space can be continuous. With the other design variable types, the design space is discontinuous.

Types of functions: The functions that are used to designate the objective and the constraint functions can be of several types. Note that the independent parameters are the design variables.

The simplest function is a *linear* function.

Nonlinear functions are those where the relationship between the function and the independent variable (e.g. design variable) is nonlinear. Note that a nonlinear function is not necessarily a polynomial. For

example, $f(x) = x^2 - 3x$ is nonlinear and so is $f(\mathbf{x}) = \frac{e^{x_1}}{\sin(x_2)}$.

Posynomial functions have a special form given by

$$f(\mathbf{x}) = C_1 x_1^{a_{11}} x_2^{a_{12}} \dots x_n^{a_{1n}} + C_2 x_1^{a_{21}} x_2^{a_{22}} \dots x_n^{a_{2n}} + \dots + C_s x_1^{a_{s1}} x_2^{a_{s2}} \dots x_n^{a_{sn}} \quad (17.2-1)$$

where $C_i > 0$, a_{ij} is a known coefficient, and $x_i > 0$.

Types of constraints: There are two types of constraints.

Equality constraints are used to express the relationship between two or more design variables using the equality operator. The function used to describe an equality constraint can be of any one of the forms defined earlier.

Inequality constraints are used to express the relationship between two or more design variables using a greater than, less than, greater than or equal to and less than or equal to operators. The function used to describe an inequality constraint can be of any one of the forms defined earlier.

Types of objective functions: There are two major types of objective functions – single objective and multi-objective functions.

Single objective. Design problems discussed in this text are driven primarily by a single, primary objective.

Multi-objective. However, there are engineering problems where more than one objective is important and needs to be minimized simultaneously. For example, an unconstrained minimization problem with multiple objectives can be stated as

$$\text{Find} \quad \mathbf{x} \in R^n \quad (17.2-2)$$

$$\text{To minimize} \quad [f_1(\mathbf{x}), f_2(\mathbf{x}), \dots, f_r(\mathbf{x})] \quad (17.2-3)$$

An example of a multi-objective structural design problem is

$$\text{Find} \quad \text{The design variables}$$

$$\text{To minimize} \quad [\text{Project Cost}, \text{Construction Time}]$$

There are two objective functions – the project cost and the construction time. In this example, it should be recognized that minimizing construction time can lead to an increase in the overall project cost.

Using the different types of objective and constraint functions, and design variables, different types of mathematical programming problems can be described.

Linear Programming (LP) Problem: Consider the following problem

$$\text{Find} \quad \{\mathbf{x}\} \quad (17.2-4a)$$

$$\text{To maximize} \quad \sum_{k=1}^n c_k x_k \quad (17.2-4b)$$

$$\text{Subject to} \quad a_{i1}x_1 + a_{i2}x_2 + \dots + a_{in} \leq b_i \quad i = 1, \dots, l \quad (17.2-4c)$$

$$a_{i1}x_1 + a_{i2}x_2 + \dots + a_{in} \geq b_i \quad i = l+1, \dots, m \quad (17.2-4d)$$

$$a_{i1}x_1 + a_{i2}x_2 + \dots + a_{in} = b_i \quad i = m+1, \dots, r \quad (17.2-4e)$$

$$x_1 \geq 0, x_2 \geq 0, \dots, x_n \geq 0 \quad (17.2-4f)$$

where the coefficients c_k and a_{ij} are constant coefficients, and b_i are fixed real constants which are required to be nonnegative. As you can see from the above problem formulation, the objective and the constraint functions are linear functions of the design variables; hence the problem is called a linear programming problem. To solve the above problem, the problem definition is transformed so that all the constraints are equality constraints and $\mathbf{b} \geq 0$. The standard LP problem is then

$$\text{Find} \quad \{\mathbf{x}\} \quad (17.2-5a)$$

$$\text{To maximize} \quad \mathbf{c}^T \mathbf{x} \quad (17.2-5b)$$

$$\text{Subject to} \quad \mathbf{A}_{r \times n} \mathbf{x}_{n \times 1} = \mathbf{b}_{r \times 1} \quad (17.2-5c)$$

$$\mathbf{x} \geq \mathbf{0} \quad (17.2-5d)$$

Plastic designs (encountered in design of steel structural systems) can under restrictive conditions, be posed as an LP problem.

Dynamic Programming (DP) Problem: DP methodology is applicable to engineering problems that can be broken into stages and exhibit Markovian³ property. While not widely used in the area of structural design, one could design a problem that could be solved effectively as a DP problem. Consider a building system consisting of a roof system, a set of floors consisting of beams and columns, and a foundation system. The load path typically starts at the roof and is finally transmitted to the foundation via the floor system starting at the top floor and progressing to the bottommost floor. Hence one could design the building in stages, starting at the roof and progressing to the foundation.

Non-Linear Programming (NLP) Problem: As we had mentioned earlier in this chapter, most engineering problems require constraints to be satisfied. These constraints, and frequently, the objective function, are nonlinear leading to an NLP problem.

$$\text{Find} \quad \mathbf{x} \in R^n \quad (17.2-6a)$$

$$\text{To minimize} \quad f(\mathbf{x}) \quad (17.2-6b)$$

$$\text{Subject to} \quad g_i(\mathbf{x}) \leq 0, \quad i = 1, 2, \dots, l \quad (17.2-6c)$$

³ A Markovian property is encompassed in a process if the decisions for optimal return at a stage in the process depend only on the current state of the system and subsequent decisions.

$$h_j(\mathbf{x}) = 0, \quad j = 1, 2, \dots, m \quad (17.2-6d)$$

$$x_k^L \leq x_k \leq x_k^U, \quad k = 1, 2, \dots, n \quad (17.2-6e)$$

Special cases of this formulation include those where the design variables are integers (**Integer Programming Problem**), or boolean (**Zero-One Programming Problem**) or discrete (**Discrete Programming Problem**). Perhaps the biggest difference between the NLP and LP problems is the likelihood of multiple solutions with NLP problems and the difficulty of finding the solution effectively.

Our focus in this chapter is to look at NLP problems. Simple NLP problems can be solved 'by hand'. However, most engineering problems must be solved numerically.

17.3 Non-Linear Programming (NLP) Problem

As we saw in the previous section, the non-linear programming problem is by far the most commonly encountered structural design problems. We will restate the nonlinear programming problem as follows.

$$\text{Find } \mathbf{x} \in R^n \quad (17.3-1a)$$

$$\text{To minimize } f(\mathbf{x}) \quad (17.3-1b)$$

$$\text{Subject to } g_i(\mathbf{x}) \leq 0, i = 1, 2, \dots, l \quad (17.3-1c)$$

$$h_j(\mathbf{x}) = 0, j = 1, 2, \dots, m \quad (17.3-1d)$$

$$x_k^L \leq x_k \leq x_k^U, k = 1, 2, \dots, n \quad (17.3-1e)$$

In addition, we will assume that (i) \mathbf{x} is continuous and real-valued, and (ii) $f(\mathbf{x})$, $g_i(\mathbf{x})$ and $h_j(\mathbf{x})$ are continuous and differentiable. The objective is to find $\mathbf{x} = \mathbf{x}^*$ so that the objective function has the lowest possible value without violating the constraints.

Mathematical Background: In Chapter 6, we examined the concept of derivatives or gradients looking at these quantities as scalar values. We must now extend this idea into multi-dimensions. The building block is a vector. For example, in two dimensional space a vector can be expressed as $\mathbf{g}_{2 \times 1} = [g_x \ g_y]^T$. An example of vector in two-dimensions in a vector of direction cosines (or, unit vector) - $\mathbf{g}_{2 \times 1} = [0.6 \ -0.8]^T$. If $f = f(\mathbf{x})$, then the gradient vector of $f(\mathbf{x})$ is written as $\nabla \mathbf{f}(\mathbf{x})$. For example, let $f(\mathbf{x}) = x_1^2 - 2x_1x_2 + x_2^3$. Then $\frac{\partial f}{\partial x_1} = 2x_1 - 2x_2$ and $\frac{\partial f}{\partial x_2} = -2x_1 + 3x_2^2$, and we can write the gradient of $f(\mathbf{x})$ as $\nabla \mathbf{f}(\mathbf{x})_{2 \times 1} = [2x_1 - 2x_2 \ -2x_1 + 3x_2^2]^T$.

Consider the problem of minimizing $f(\mathbf{x})$ subject to $h_j(\mathbf{x}) = 0$. A point \mathbf{x}^* is a *regular point* provided $\mathbf{h}(\mathbf{x}^*) = 0$ and the gradients of all the constraints at \mathbf{x}^* are linearly independent. The linear independence arises from the fact that no two gradients are parallel to each other nor is it possible to write a gradient as a linear combination of two or more of the other gradients. For example, let $h_1(x_1, x_2) = 2x_1^2 - 3x_2$ and $h_2(x_1, x_2) = -12x_1^2 + 18x_2$. Then $\nabla \mathbf{h}_1 = [4x_1 \ -3]^T$ and $\nabla \mathbf{h}_2 = [-24x_1 \ 18]^T$. Let $x^* = \{2, 1\}$. Then $\nabla \mathbf{h}_1(\mathbf{x}^*) = [4 \ -3]^T$ and $\nabla \mathbf{h}_2(\mathbf{x}^*) = [-24 \ 18]^T$. The two vectors $\nabla \mathbf{h}_1$ and $\nabla \mathbf{h}_2$ are parallel (or, linearly dependent) to each other since $\nabla \mathbf{h}_2 = -6\nabla \mathbf{h}_1$.

17.3.1 KUHN-TUCKER CONDITIONS

Let us consider the following problem.

$$\text{Find } \mathbf{x} \in R^n \quad (17.3.1-1a)$$

$$\text{To minimize } f(\mathbf{x}) \quad (17.3.1-1b)$$

$$\text{Subject to } h_j(\mathbf{x}) = 0, j = 1, 2, \dots, m \quad (17.3.1-1c)$$

Lagrange is credited with developing a simple but effective way of solving the problem. He suggested that a function L (called the Lagrangian) be developed as

$$L = f(\mathbf{x}) + \sum_{j=1}^m \lambda_j h_j(\mathbf{x}) \quad (17.3.1-2)$$

where λ_j is called the Lagrange multiplier. The regular point \mathbf{x}^* is a stationary point if

$$\frac{\partial L}{\partial x_i} = 0 = \frac{\partial f}{\partial x_i} + \sum_{j=1}^m \lambda_j \frac{\partial h_j(\mathbf{x})}{\partial x_i} \quad i = 1, 2, \dots, n \quad (17.3.1-3)$$

$$\text{and } h_j(\mathbf{x}) = 0 \quad j = 1, 2, \dots, m \quad (17.3.1-4)$$

In other words, solving Eqns. (17.3.1-3) and (17.3.1-4) is equivalent to solving the original problem given by Eqns. (17.3.1-1a)-(17.3.1-1c)). Note that there are $(m + n)$ unknowns in these $(m + n)$ equations. However, the form of these equations is dependent on the form of the objective function and the equality constraints and the equations are usually not linear equations. These conditions are known as first-order necessary conditions. The second-order sufficient conditions establish whether \mathbf{x}^* is a local minimum or not. Treatment of these second-order conditions is outside the scope of this text⁴.

⁴ See a book on optimal design. A comprehensive list is provided in the Bibliography at the end of the text.

Example 17.3-1 Constrained Minimization with Equality Constraint

Find $\{x_1, x_2\}$

To minimize $4x_1 - x_2$

Subject to $2x_1^2 + x_2^2 = 1$

Solution

Step 1: Form the Lagrangian

$$L = 4x_1 - x_2 + \lambda_1(2x_1^2 + x_2^2 - 1)$$

Step 2: Using Eqn. (17.3.1-3)

$$\frac{\partial L}{\partial x_1} = 0 = 4 - 4\lambda_1 x_1 \Rightarrow x_1 = \frac{1}{\lambda_1}$$

$$\frac{\partial L}{\partial x_2} = 0 = -1 + 2\lambda_1 x_2 \Rightarrow x_2 = \frac{1}{2\lambda_1}$$

Substituting in Eqn. (17.3.1-4) or the constraint equation, we have

$$2\left(\frac{1}{\lambda_1}\right)^2 + \left(\frac{1}{2\lambda_1}\right)^2 = 1$$

Solving, $\lambda_1 = \pm \frac{3}{2}$. For each root, we have

$$\lambda_1 = \frac{3}{2} : x_1^* = \frac{2}{3} \text{ and } x_2^* = \frac{1}{3}. f(\mathbf{x}^*) = \frac{7}{3}$$

$$\lambda_1 = -\frac{3}{2} : x_1^* = -\frac{2}{3} \text{ and } x_2^* = -\frac{1}{3}. f(\mathbf{x}^*) = -\frac{7}{3}$$

Obviously the minimum is at $x_1^* = -\frac{2}{3}$, $x_2^* = -\frac{1}{3}$ and $f(\mathbf{x}^*) = -\frac{7}{3}$.

Now let us consider the following problem with equality and inequality constraints.

$$\text{Find } \mathbf{x} \in R^n \quad (17.3.1-5a)$$

$$\text{To minimize } f(\mathbf{x}) \quad (17.3.1-5b)$$

$$\text{Subject to } h_j(\mathbf{x}) = 0, j = 1, 2, \dots, m \quad (17.3.1-5c)$$

$$g_i(\mathbf{x}) \leq 0, i = 1, 2, \dots, l \quad (17.3.1-5d)$$

The Lagrangian function for this problem can be defined as

$$L = f(\mathbf{x}) + \sum_{j=1}^m \lambda_j h_j(\mathbf{x}) + \sum_{i=1}^l \mu_i g_i(\mathbf{x}) \quad (17.3.1-6)$$

where λ_j and μ_i are the Lagrange multipliers. The regular point \mathbf{x}^* is a stationary point if

$$\frac{\partial L}{\partial x_k} = 0 = \frac{\partial f}{\partial x_k} + \sum_{j=1}^m \lambda_j \frac{\partial h_j(\mathbf{x})}{\partial x_k} + \sum_{i=1}^l \mu_i \frac{\partial g_i(\mathbf{x})}{\partial x_k} \quad k = 1, 2, \dots, n \quad (17.3.1-7)$$

$$h_j(\mathbf{x}) = 0 \quad j = 1, 2, \dots, m \quad (17.3.1-8)$$

$$g_i(\mathbf{x}) \leq 0 \quad i = 1, 2, \dots, l \quad (17.3.1-9)$$

$$\mu_i g_i(\mathbf{x}) = 0 \quad i = 1, 2, \dots, l \quad (17.3.1-10)$$

$$\mu_i \geq 0 \quad i = 1, 2, \dots, l \quad (17.3.1-11)$$

In other words, solving Eqns. (17.3.1-7)-(17.3.1-11) is equivalent to solving the original problem given by Eqns. (17.3.1-5a)-(17.3.1-5d). We will illustrate the usage of these conditions using an example.

Example 17.3-2 Constrained Minimization with inequality constraint

Find $\{x_1, x_2\}$

To minimize $(x_1 - 3)^2 + (x_2 - 4)^2$

Subject to $x_1 + x_2 - 5 \leq 0$

$x_1 \geq 0, x_2 \geq 0$

Solution

Step 1: We will not use the last two constraints requiring that the design variable be positive but will use them to select the optimal solution. Form the Lagrangian as

$$L = (x_1 - 3)^2 + (x_2 - 4)^2 + \mu_1(x_1 + x_2 - 5)$$

Step 2: Hence, the necessary conditions to be satisfied are

$$\frac{\partial L}{\partial x_1} = 0 = 2x_1 - 6 + \mu_1 \quad (1)$$

$$\frac{\partial L}{\partial x_2} = 0 = 2x_2 - 8 + \mu_1 \quad (2)$$

$$\mu_1(x_1 + x_2 - 5) = 0 \quad (3)$$

$$x_1 + x_2 - 5 \leq 0 \quad (4)$$

$$\mu_1 \geq 0 \quad (5)$$

Step 3: The key to solving these conditions is to recognize the importance of Eqn. (3). Eqn. (17.3.1-10) represents the switching conditions since from those equations either $\mu_i = 0$ or $g_i = 0$. Using the switching conditions, the possibilities for this problem are discussed below.

Case 1: $\mu_1 = 0$ (meaning that the inequality constraint, g_1 is not active)

From (1) and (2), $2x_1 - 6 = 0$ and $2x_2 - 8 = 0$, yielding $x_1 = 3$ and $x_2 = 4$. However, this solution does not satisfy Eqn. (4). Hence this is an unacceptable case.

Case 2: $g_1 = 0$ (meaning that the inequality constraint, g_1 is active)

Hence,

$$2x_1 - 6 + \mu_1 = 0$$

$$2x_2 - 8 + \mu_1 = 0$$

$$x_1 + x_2 - 5 = 0$$

Solving these linear simultaneous equations, $x_1 = 2$, $x_2 = 3$ and $\mu_1 = 2$. These values satisfy Eqns. (4), (5) and the requirements that $x_1 \geq 0, x_2 \geq 0$. Hence $\mathbf{x}^* = \{2, 3\}$. For these values, the objective function is $f(\mathbf{x}^*) = 2$.

Example 17.3-3 Constrained Minimization with inequality constraints

Find $\{x_1, x_2\}$

To maximize $-4x_1^2 - 2x_2$

Subject to $-2x_1 + x_2 \leq 4$

$$x_1 + 2x_2 \geq 2$$

$$x_1 \geq 0, x_2 \geq 0$$

Solution

Step 1: First thing to notice about the problem is that it is not in the standard form. We must first convert the problem to a minimization problem. Maximizing $f(\mathbf{x})$ is equivalent to minimizing $-f(\mathbf{x})$. Hence, $f(\mathbf{x}) = 4x_1^2 + 2x_2$. Second, the second constraint must be transformed to a ≤ 0 constraint. Note that $g(\mathbf{x}) \geq a$ is equivalent to $-g(\mathbf{x}) \leq -a$. Hence $g_2 \equiv -x_1 - 2x_2 + 2 \leq 0$. As with the previous problem, we will not use the last two constraints requiring that the design variable be positive but will use them to select the optimal solution. Form the Lagrangian as

$$L = 4x_1^2 + 2x_2 + \mu_1(-2x_1 + x_2 - 4) + \mu_2(-x_1 - 2x_2 + 2)$$

Step 2: Hence, the necessary conditions to be satisfied are

$$\frac{\partial L}{\partial x_1} = 0 = 8x_1 - 2\mu_1x_1 - \mu_2 \quad (1)$$

$$\frac{\partial L}{\partial x_2} = 0 = 2 + \mu_1 - 2\mu_2 \quad (2)$$

$$\mu_1(-2x_1 + x_2 - 4) = 0 \quad (3)$$

$$\mu_2(-x_1 - 2x_2 + 2) = 0 \quad (4)$$

$$-2x_1 + x_2 - 4 \leq 0 \quad (5)$$

$$-x_1 - 2x_2 + 2 \leq 0 \quad (6)$$

$$\mu_1 \geq 0, \mu_2 \geq 0 \quad (7)$$

Step 3: Using the switching conditions, there are four possibilities.

Case 1: $\mu_1 > 0, \mu_2 > 0$ (meaning that the inequality constraints are active, $g_1 = 0$ and $g_2 = 0$)

Hence, $-2x_1 + x_2 - 4 = 0$

$$-x_1 - 2x_2 + 2 = 0$$

$$8x_1 - 2\mu_1 x_1 - \mu_2 = 0$$

$$2 + \mu_1 - 2\mu_2 = 0$$

From the first two equations, $x_1 = -6/5$. This is an unacceptable solution.

Case 2: $\mu_1 > 0, \mu_2 = 0$ (meaning that the inequality constraint $g_1 = 0$)

Hence, $-2x_1 + x_2 - 4 = 0$

$$8x_1 - 2\mu_1 x_1 = 0$$

$$2 + \mu_1 = 0$$

From the last equation, $\mu_1 = -2 < 0$, and is an unacceptable solution.

Case 3: $\mu_1 = 0, \mu_2 = 0$ (meaning that the inequality constraints are inactive)

$$8x_1 = 0$$

$$2 = 0$$

The second equation expresses an invalid condition, and the solution is unacceptable.

Case 4: $\mu_1 = 0$, $\mu_2 > 0$ (meaning that the inequality constraint $g_2 = 0$)

Hence, $-x_1 - 2x_2 + 2 = 0$

$$8x_1 - \mu_2 = 0$$

$$2 - 2\mu_2 = 0$$

Solving, $x_1 = 1/8$, $x_2 = 15/16$ and $\mu_2 = 1$. The constraint g_1 is satisfied. Hence, $\mathbf{x}^* = \left\{ \frac{1}{8}, \frac{15}{16} \right\}$ and $f(\mathbf{x}^*) = 1.9375$.

The two examples illustrate the strengths and weaknesses of the Kuhn-Tucker (K-T) conditions. Now, a few notes about the first-order K-T conditions.

- (a) A regular point is a candidate K-T point. In other words, a non-regular point cannot be used to satisfy the K-T conditions.
- (b) Points that satisfy K-T conditions are local minimum points. However, all local minimum points are not K-T points. In other words, K-T conditions may not be able to locate all local minimum points.
- (c) It may not be possible to obtain an analytical solution to K-T conditions. This is because the resulting equations may be nonlinear equations requiring a numerical method.

EXERCISES

Appetizers

Problem 17.3-1

An aluminum can manufacturer needs to design a (closed) cylindrical soft drink can. The volume of the can must be 400 ml . The height, h of the can must be at least twice the diameter d and cannot be more than thrice the diameter. Packaging considerations restrict the height to 25 cm and the usage of the least amount of sheet metal. Formulate the optimal design problem by clearly identifying the design variables, objective function and constraints.

Problem 17.3-2

A box frame is made of steel angle and hollow tube sections as shown in the Fig. P17.3-2. It must be designed for least cost. The angle sections cost \$200/m and the tube sections cost \$500/m. The box frame must enclose a space of 1000 m^3 . A side of the box cannot be less than 2 m. Formulate the optimal design problem by clearly identifying the design variables, objective function and constraints.

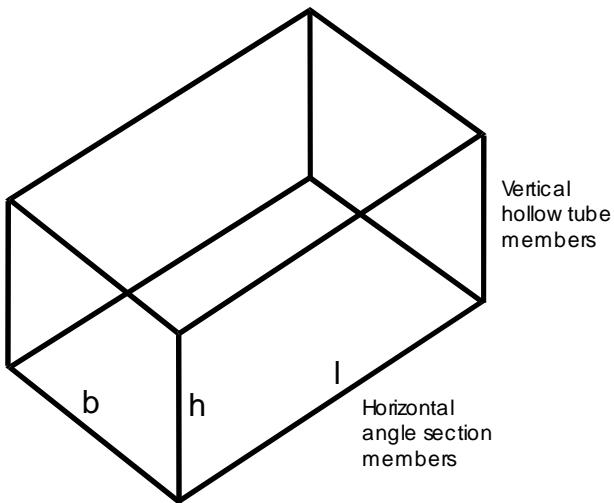


Fig. P17.3-2

Problem 17.3-3

A factory makes two products labeled A and B . The manufacturing of these products takes place in two stages – Stage One and Stage Two. Departments 1 and 2 handle Stage One and Stage Two respectively. Product A requires 2 hours for Stage One and 2.25 hours for Stage Two. Product B requires 2 hour for Stage One and 1.75 hours for Stage Two. Each of the two Departments can be operational for a total of 20 hours per day, seven days a week. The profit from sale of product A is \$1.35 per unit and for product B is \$1.2 per unit. How many units of A and B should be produced per week so as to maximize the profit?

Main Course

Problem 17.3-4

Solve *Problem 17.3-1* using the graphical technique. Verify your answer using K-T conditions.

Problem 17.3-5

Solve *Problem 17.3-2* using the K-T conditions.

Problem 17.3-6

Solve *Problem 17.3-3* using the graphical technique. Verify your answer using K-T conditions.

Structural Concepts

Problem 17.3-7

Answer *True* or *False*. If *False* state the reason(s) why.

- (a) A feasible design is the one with the lowest objective function value.
- (b) An optimal design problem must have constraints.
- (c) The number of inequality constraints in a problem cannot be greater than the number of design variables for a problem to be well-posed.
- (d) Every optimal design problem has one or more optimum solutions.
- (e) An inequality constraint $h(\mathbf{x}) = 0$ can be expressed as two inequality constraints $h(\mathbf{x}) \leq 0$ and $h(\mathbf{x}) \geq 0$.

17.3.2 NUMERICAL SOLUTION TECHNIQUES

As you have seen earlier on in this book, problems such as root finding or solution of linear algebraic equations can be carried out by more than one solution technique. Different techniques have different assumptions, restrictions, strengths and weaknesses. Similarly, there are several numerical techniques that can potentially be used to solve NLP problems. They are broadly classified as either gradient-based techniques or direct-search techniques.

Gradient-Based Techniques: The nonlinear nature of the problem makes it difficult to solve the problem directly. Instead the solution is obtained iteratively. The original nonlinear problem is transformed into a more manageable subproblem whose solution can be obtained numerically. Solution to these subproblems requires that not only the function values be known but also that their gradients (or, derivatives) with respect to the design variables be known. Some of the popular solution techniques are Sequential Linear Programming (SLP) technique, Sequential Quadratic Programming (SQP) technique, Feasible Directions Method, Generalized Reduced Gradient (GRG) Method, Augmented Lagrangian technique, Sequential Unconstrained Minimization Technique (SUMT) etc. Treatment of one or more of these techniques can be found in any book on optimization techniques.

Direct Search Techniques: The direct search techniques do not require derivatives. Hence they have the advantage of being able to solve not only problems where the derivatives are discontinuous but also have the ability to find the global minimum. This advantage is offset with an increase in computational requirement – usually the function values are required at a very large number of locations in the design space. Some of popular solution techniques are Hooke and Jeeves Method, Powell's Method of Conjugate Directions, Simulated Annealing (SA), Genetic Algorithm (GA) etc.

In the next section we will look at the Genetic Algorithm in some detail and understand how to use the method for solving constrained minimization problems.

17.4 Genetic Algorithm

Genetic Algorithm (GA) is a search strategy based on the rules of natural genetic evolution. Before the traits of genetic systems were used in solving optimization problems, biologists had used digital computers to perform simulations of genetic system as early as the early 50's. The application of genetic algorithms for adaptive systems was first proposed by John Holland (University of Michigan) in 1962, and the term "Genetic Algorithms" was first used in his student's dissertation.

GA's have been used to solve a variety of structural design problems. They include the optimal design of all the structural systems that we have seen in this text and more. Because of their discrete nature, GA's lend themselves well to the process of automating the design of skeletal structures. GA's do not require gradient or derivative information. For this reason alone, it has been applied by researchers to solve discrete, non-differentiable, combinatory and global optimization engineering problems, such as transient optimization of gas pipeline, topology design of general elastic mechanical system, time scheduling, circuit layout design, composite panel design, pipe network optimization, and several hundred others. GA's are recognized to be different than traditional gradient-based optimization techniques in the following four major ways [Goldberg, 1989].

1. GA's work with a coding of the design variables and parameters in the problem, rather than with the actual parameters themselves.
2. GA's make use of population-type search. Many different design points are evaluated during each iteration instead of sequentially moving from one point to the next.
3. GA's need only a fitness or objective function value. No derivatives or gradients are necessary.
4. GA's use probabilistic transition rules to find new design points for exploration rather than using deterministic rules based on gradient information to find these new points.

The idea behind GA is to simulate the behavior of natural evolutionary selection. Although there exist many different variations of GA's, the basic structure is the same as shown in Fig. 17.4-1.

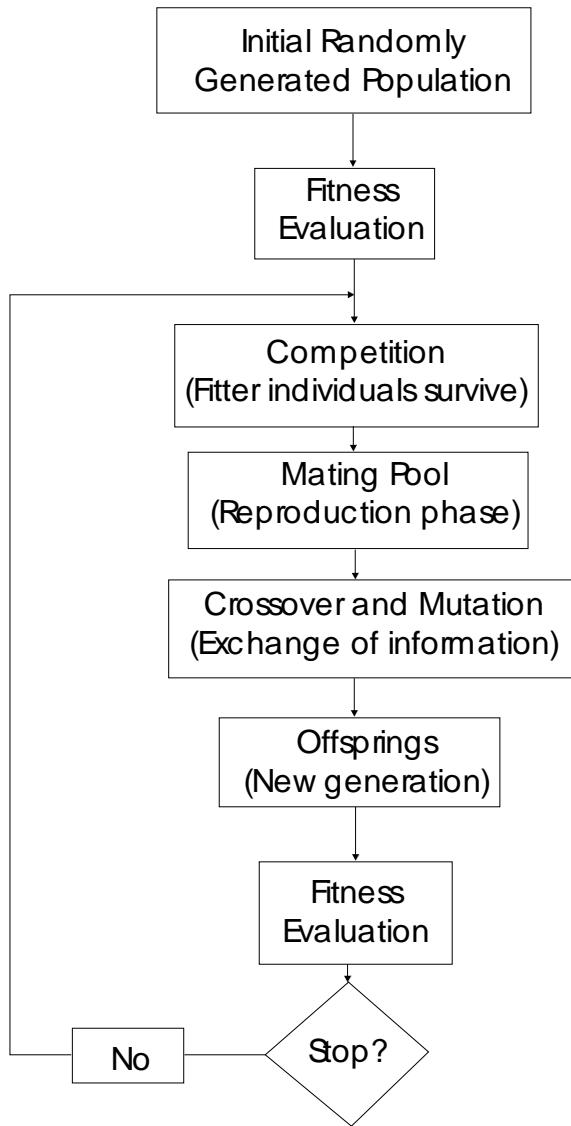


Fig. 17.4-1 Flow in a Simple Genetic Algorithm (SGA)

We will explain the basic flow shown in Fig. 17.4-1 next.

17.4.1 THE BASIC ALGORITHM

The genetic algorithm is used to solve the following problem.

$$\text{Minimize} \quad \hat{f}(\mathbf{x}) \quad (17.4.1-1a)$$

$$\text{Subject to} \quad x_k^L \leq x_k \leq x_k^U, \quad k = 1, 2, \dots, n \quad (17.4.1-1b)$$

Note that the problem is primarily an unconstrained minimization problem with lower and upper bounds on the design variables. We will first present the necessary background before detailing the algorithm. In the language of GA, we will be computing $\hat{f}(\mathbf{x})$, the fitness function, not $f(\mathbf{x})$, the objective function. The two functions are related and the distinction between the two will be made later.

Binary Encoding and Decoding of Design Variables

Binary encoding is the most popular way of encoding the design variables. A binary number is represented as $(b_m \dots b_1 b_0)_2$ where b_i is either 0 or 1. For example, $(101)_2$ is a three-digit or three-bit binary number. To understand this concept we will first explain the relationship between binary and decimal numbers.

$$(b_m \dots b_1 b_0)_2 = (2^0 b_0 + 2^1 b_1 + \dots + 2^m b_m)_{10} \quad (17.4.1-2)$$

Hence, as an example

$$(101)_2 = (2^0 \cdot 1 + 2^1 \cdot 0 + 2^2 \cdot 1)_{10} = (5)_{10}$$

In other words, the binary number $(101)_2$ is equal to decimal 5.

The process of taking a decimal number and constructing its binary representation (not value) is called encoding. Decoding is the inverse process of taking the binary encoded value and constructing its decimal equivalent.

Continuous Design Variables: A design variable x_i is between x_i^L and x_i^U . Note that x_i is a decimal number. If m bits are available to represent x_i , then the precision p_i with which the number is represented is given by

$$p_i = \frac{x_i^U - x_i^L}{2^m - 1} \quad (17.4.1-3)$$

To understand the term in the denominator, let us look at the example with 3 bits. The possible binary representations with 3 bits are 000, 001, 010, 011, 100, 101, 110 and 111. Or, 8 possible combinations. Similarly, with 4 bits we have 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110 and 1111. Or, 16 possible combinations. In other words, if there are m bits then there are 2^m combinations or $2^m - 1$ intervals. The range of values between x_i^L and x_i^U is divided into $2^m - 1$ intervals.

For example, if $x^L = 1$, $x^U = 8$ and $m = 3$, then $p = \frac{8-1}{7} = 1$. The following table shows the relationship between the binary representation and their decimal equivalents with this example.

Binary Representation	Decimal Equivalent
$(b_2 b_1 b_0)_2$	$x = x^L + p(b_2 b_1 b_0)_2$
000	1.0
001	2.0
010	3.0
011	4.0
100	5.0
101	6.0
110	7.0
111	17.0

The decoding is achieved using

$$x = x^L + p(b_m \dots b_1 b_0)_2 \quad (17.4.1-4)$$

where decimal values are used. The previous example is quite straightforward since the bounds and the precision are integers. Now consider the following example. Let $x^L = 1$, $x^U = 10$ and $m = 3$. Then $p = \frac{10 - 1}{7} = 1.28571$ if we assume that the precision is finite. The new relationship is as follows.

Binary Representation	Decimal Equivalent
$(b_2 b_1 b_0)_2$	$x = x^L + p(b_2 b_1 b_0)_2$
000	1.0
001	2.25571
010	3.57143
011	4.85714
100	6.14286

101	7.42857
110	8.71429
111	10.0

Integer Design Variable: The above example illustrates the encoding and decoding problems when the range is not a multiple of $2^m - 1$. With integer design variables, one approach is to apply Eqn. (17.4.1-3) with the precision p being 1 and compute the least number of bits required to achieve the precision. The number of bits obviously is an integer. For example, let $x^L = 1$, $x^U = 10$ and $p = 1$. Using

$$2^m - 1 = \frac{x_i^U - x_i^L}{p} \Rightarrow m = \log(x_i^U - x_i^L + 1)/\log(2) \quad (17.4.1-5)$$

we have $m = \log(10 - 1 + 1)/\log(2) = 3.32193$. We will round 3.32193 to the next highest integer, 4.

Hence, the new precision with 4 bits is $p = \frac{10 - 1}{2^4 - 1} = 0.6$. Once we compute the decimal equivalent, we can either truncate or round the value to an integer.

Binary Representation $(b_3 b_2 b_1 b_0)_2$	Decimal Equivalent $x = x^L + p(b_3 b_2 b_1 b_0)_2$	Integer Equivalent (rounded value)
0000	1.0	1
0001	1.6	2
0010	2.2	2
0011	2.8	3
0100	3.4	3
0101	4.0	4
0110	4.6	5
0111	5.2	5
1000	5.8	6

1001	6.4	6
1010	7.0	7
1011	7.6	8
1100	8.2	8
1101	8.8	9
1110	9.4	9
1111	10.0	10

While problems exist with the procedure (note that some integers - 2,3,5,6,8,9 appear more than once), experience has shown that the procedure works quite well for most problems.

Discrete Design Variable: The representation is similar to integer design variables with $x^L = 1$ and $x^U = q$ where there are q possible discrete values. The discrete values are usually stored in a table in some sorted manner and the integer value between 1 and q is used as an index to obtain the corresponding value(s) from the table.

Zero-One (Binary) Design Variable: There is nothing special that needs to be done since we need exactly one bit to represent a zero-one design variable.

Chromosome: To represent all the design variables in a problem, we need to create the *chromosome* for the problem. A chromosome is a concatenated binary string of all the binary representations of the design variables. If there are n design variables with $m = 3$ to represent each design variable, then the chromosome looks as shown in Fig. 17.4.1-1 with X being 0 or 1.

XXX XXX XXX XXX
 X₁ X₂ X₃ X_n

Fig. 17.4.1-1 Possible chromosome (or, gene)

The number of bits do not have to be equal for all the design variables nor do the design variables have to be ordered from 1 to n in the chromosome.

The basic steps in the algorithm are discussed next.

Initial Population: The first step is to create the initial population. Unlike gradient-based methods where the search for the optimal solution takes place by moving from one point to the next, in a GA the traits of a population (of members) are used to move from one generation to the next. Fig. 17.4.1-2 shows an initial population consisting of z members. The initial population is usually created randomly.

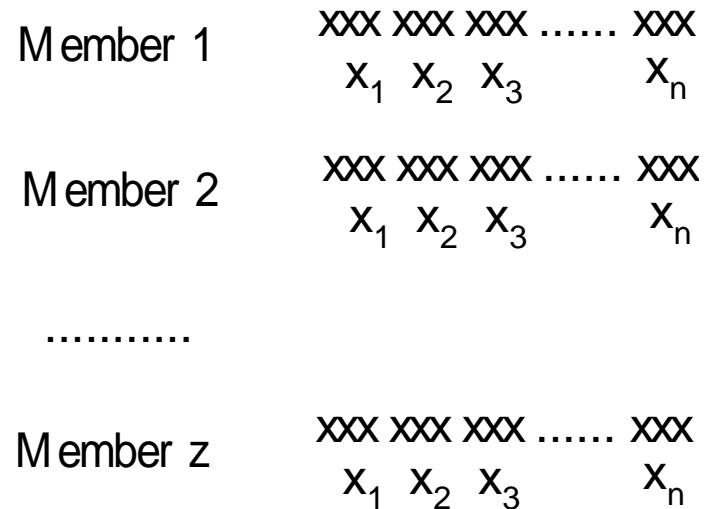


Fig. 17.4.1-2 Initial population

With the example in Fig. 17.4.1-2, the size of the chromosome is $3n$ bits. A random number generator can be used to generate a random number between 0.0 and 1.0. Invoking the random number generator $3n$ times, we can generate each member of the population as follows – if the random number is ≤ 0.5 then a 0 is assigned to that bit otherwise if the number is > 0.5 a 1 is assigned to that bit.

We will study the effect of the size of the population, z at the end of this section.

Fitness Evaluation

Once the initial population is generated, the actual search process starts. The chromosome is decoded to obtain the values of the design variables, \mathbf{x} and the fitness function value is computed for each member of the population. In other words, there are z fitness values $\hat{f}(\mathbf{x})$ that are calculated.

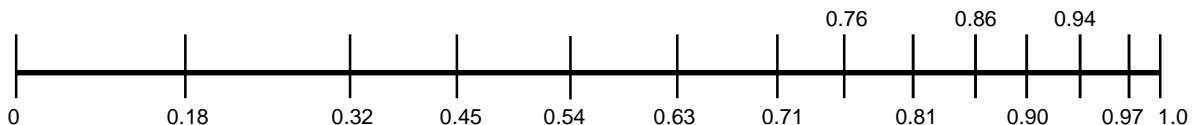
Reproduction

To generate the members of the next generation, the reproduction phase has at least three distinct steps. First the mating pool is created. Typically, the weaker members (higher fitness values) are replaced with stronger members (lower fitness values). To produce offspring, two members from the mating pool are selected and a crossover operation is carried out to create the chromosome of the offspring. Finally, to bring diversity into the population, the mutation operation is carried out.

Mating Pool: The mating pool is constructed by selecting members from the population. We will describe two commonly used methods. In the *roulette wheel selection*, the chance of being selected is based on the fitness value. The individual members of the population are mapped to segments of a line such that the length of the segment is related to its fitness value.

Individual	1	2	3	4	5	6	7	8	9	10	11	12	13	Sum
Fitness	1.20	1.50	1.70	2.20	2.50	2.70	3.90	4.50	4.70	5.20	5.50	6.10	7.80	49.50
	41.25	33.00	29.12	22.50	19.80	18.33	12.69	11.00	10.53	9.52	9.00	8.11	6.35	231.21
Selection Probability	0.18	0.14	0.13	0.10	0.09	0.08	0.05	0.05	0.05	0.04	0.04	0.04	0.03	1.00
Cumulative Value	0.18	0.32	0.45	0.54	0.63	0.71	0.76	0.81	0.86	0.90	0.94	0.97	1.00	

The sum of the fitness values is $S = \sum_{i=1}^{13} \hat{f}_i = 49.5$. A scaled fitness value⁵ is created as $\hat{f}_{is} = \frac{S}{\hat{f}_i}$. Let $S_S = \sum_{i=1}^{13} \hat{f}_{is} = 231.21$. The selection probability, $p_i = \frac{\hat{f}_{is}}{S_S}$. As can be seen from the table, the length of the segment is more for lower fitness values than for larger fitness values.



For selecting the individual into the mating pool, a random number between 0 and 1.0 is generated. For example, if 7 random numbers are generated as $\{0.79, 0.10, 0.33, 0.01, 0.99, 0.51, 0.83\}$, then the individuals selected are (8,1,3,1,13,4,9).

In the *tournament selection method*, using a random number generator, two members of the population are selected. Their fitness values are compared head-to-head and the one with the lower fitness value is put into the mating pool. This is done z times to create the mating pool of size z . In a “double elimination” tournament selection method, all the individuals in the population are placed in a bag. Two individuals are chosen at random. Their fitness values are compared head-to-head and the one with the lower fitness value is put into the mating pool. These two individuals are then eliminated from the bag and the process is repeated until the bag is empty. This will occur when the mating pool is half full. To complete the mating pool, the process is repeated once again.

In a simple GA, once the mating pool is constructed, two parents are selected and the reproduction process is carried out using the crossover and mutation operators.

Crossover: There are several types of crossover operators. We will illustrate three most commonly discussed operators.

One-point crossover. Consider two chromosomes selected randomly from the mating pool. They are labeled Parent 1 and Parent 2 in Fig. 17.4.1-3.

⁵ To generate a general procedure the fitness values are transformed, if necessary, so that all the values are greater than zero.

Parent 1 10001001

Parent 2 00110111

Fig. 17.4.1-3 Parents selected for the crossover operation

Based on a predetermined probability a single crossover point is chosen. If the length of the chromosome is n_c bits, then a random number is generated between 1 and n_c . This point or location is used as the crossover point. Two offspring are formed and they become the part of the next generation. The first offspring is formed by taking the front or left section of Parent 1 and the rear or right section of Parent 2. The second offspring is formed by taking the front or left section of Parent 2 and the rear or right section of Parent 1. The results are shown in Fig. 17.4.1-4.

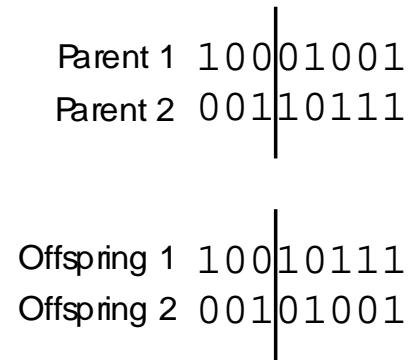


Fig. 17.4.1-4 Offspring resulting from one-point crossover operation occurring at location 3

Two-point crossover. The idea of the single point crossover can be extended to include multi-point crossover locations. The section between the first variable and the first crossover point is not exchanged. However, the bits between every other successive crossover point are exchanged between the two parents. This process is illustrated with a two-point crossover example.

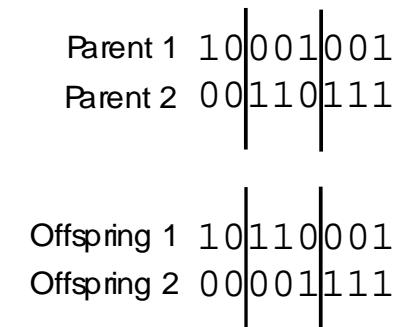


Fig. 17.4.1-5 Offspring resulting from two-point crossover operation occurring at locations 2 and 5

Uniform crossover. In uniform crossover, every location is a potential crossover point. First, a crossover mask is created randomly. This mask has the same length as the chromosome and the bit

value (parity) is used to select which parent will supply the offspring with the bit. If the mask value is 0 then the bit is taken from the first parent; otherwise the bit is taken from parent 2.

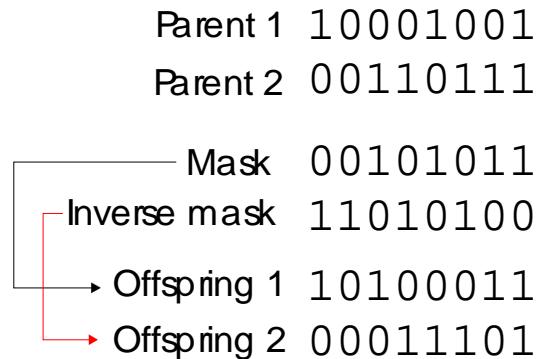


Fig. 17.4.1-6 Example showing uniform crossover

If two offspring are needed, the mask is used with the parents to create the first offspring and the inverse of the mask is used to create the second offspring.

Mutation: This operator occurs much less frequently both in nature and in GA. Offspring variables are mutated by the small random changes with a low probability. The basic idea is to introduce some diversity into the population. In other words, delay the situation where all the population becomes so homogenous that no further improvement is possible. If the length of the chromosome is n_c bits, then a random number is generated between 1 and n_c . The bit at that location is switched. An example is shown in Fig. 17.4.1-7.

Before	10010111
After	10000111

Fig. 17.4.1-7 Example showing mutation taking place at location 4

Next Generation

The new generation is formed when sufficient offspring are generated in the reproduction phase. The whole process of fitness evaluation and reproduction starts all over again with this new population. Obviously, somewhere along the evolutionary procedure the iterative process is stopped. Typically this is done if a predetermined number of iterations have been completed or if the fitness function does not change appreciably. Unlike most gradient-based techniques, there is no convergence criterion for the iterative process associated with the GA.

17.4.2 PROBLEM FORMULATION

GA's were developed to tackle unconstrained optimization problems. However, as was mentioned before, most engineering and structural design problems are constrained optimization problems. The standard approach is to transform the original constrained problem to an unconstrained problem as follows.

Find \mathbf{x}

$$\text{To minimize } \hat{f}(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^l c_i \cdot \max(0, g_i) + \sum_{j=1}^m c_j \cdot |h_j| \quad (17.4.2-1)$$

$$\text{Subject to } x_k^L \leq x_k \leq x_k^U, k = 1, 2, \dots, n$$

where c_i and c_j are penalty parameters. The selection of appropriate penalty weights c_i and c_j is always problematic even in traditional NLP schemes. Typically, a large value is used initially for these parameters. These values are then reduced as the design iterations continue. This feature is implemented in the UNDO[©] program.

17.5 Design Examples

In this section we solve several design problems using the Genetic Algorithm as implemented in the UNDO[©] program⁶. Default values are chosen for the parameters associated with the GA with the option of changing these values if required. The GA parameters under the control of the user are shown in Fig. 17.5-1.

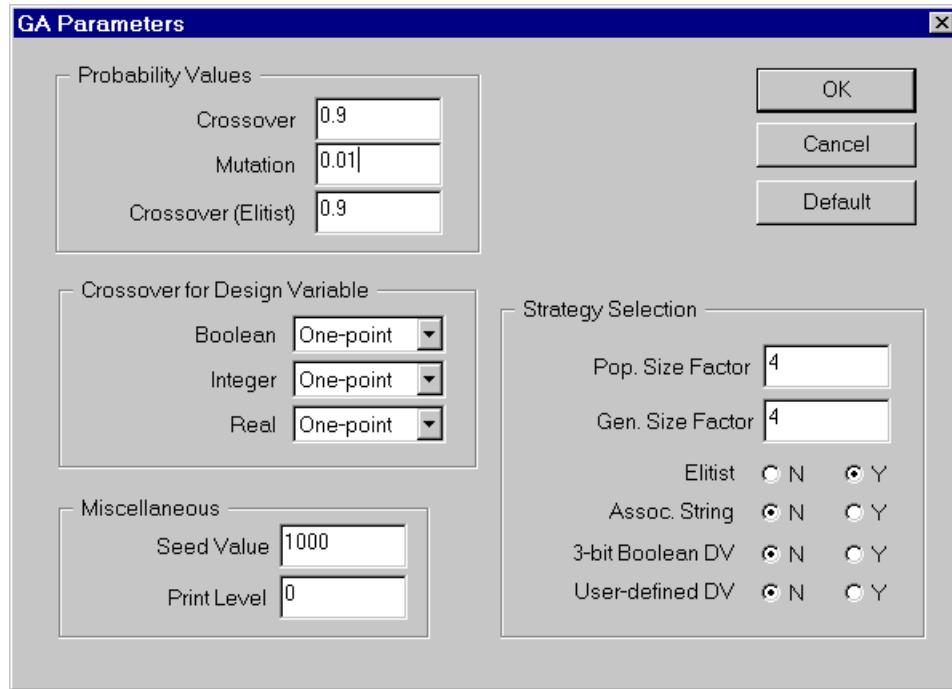


Fig. 17.5-1 User-controllable GA parameters

Probability Values

Crossover: The probability value used to determine if crossover should or should not take place.

Mutation: The probability value used to determine if mutation should or should not take place.

Crossover (Elitist): The probability value used to determine if crossover should or should not take place if the elitist strategy is used.

⁶ The UNDO[©] Tutorial and User's Manual is in the **manual** subdirectory of the directory where the program is installed.

Crossover for Design Variables

There are three types of crossovers – one point, two point and uniform.

Boolean: Type of crossover for boolean (zero-one) design variables.

Integer (including discrete): Type of crossover for integer design variables.

Float: Type of crossover for floating (or, real/continuous) design variables.

Strategy Selection

Elitist: The best member of current population is carried over to the next generation.

Association String: The program internally tries to find the association between different design variables by using an association string.

3-Bit Design Variables: Three bits are used to store the representation of boolean design variables.

User-Defined DV: Initial guess provided by the user is used to create a member of the initial population.

Miscellaneous Values

Seed Value: Seed value used in random number generation.

Print Code: Use a nonzero value so that the program creates output text files (GA_optm.out and GA_optm.hst) that contain additional GA-related output.

We will present a few guidelines for formulating and solving problems using GA's.

- (1) It is a good idea to start solving a problem with as few design variables as possible. It is easier to debug the problem formulation with a manageable number of design variables.
- (2) The selection of the lower and upper bounds must be done with care. It is necessary to have some prior knowledge of the possible range of values that the design variables can assume. One approach is to start with a wide range and obtain the solution. Once a solution is obtained, one can reduce the range by increasing the lower bound or decreasing the upper bound or both.
- (3) The penalty approach to handling constrained optimization problems works best if the constraints are normalized. For example, consider a problem where $0 \leq x_1 \leq 10$ and $-10 \leq x_2 \leq 5$. Instead of writing the following two constraints as

$$g_1(\mathbf{x}) = x_1^2 - 4x_2^3 + 12000 \leq 0$$

$$g_2(\mathbf{x}) = 40x_1x_2^2 - \frac{1000}{x_2 + 20} \leq 0$$

one can rewrite them as

$$g_1(\mathbf{x}) = \frac{x_1^2 - 4x_2^3}{12000} + 1 \leq 0$$

$$g_2(\mathbf{x}) = \frac{(x_1x_2^2)(x_2 + 20)}{10^4} - \frac{1}{400} \leq 0$$

The basic idea is to avoid very large positive and negative values.

- (4) Avoid using equality constraints. More often than not, equality constraints can be rewritten expressing one design variable in terms of the others. In other words, a design variable can be eliminated from the problem. Consider a problem where an equality constraint is

$$24x_3 - 4x_4 + 36 = 0$$

The constraint can be rewritten as

$$x_4 = 6x_3 + 9$$

and x_4 can be eliminated as a problem parameter.

- (5) Changing the default GA parameters: For those occasions when the GA does not lead to a feasible solution or leads to an unsatisfactory solution, it may be worthwhile changing the default values of the GA parameters.

Example 17.5-1 Constrained Minimization (Box Design Problem)

Find the optimal solution to the following problem.

$$\text{Find } \{x_1, x_2, x_3\}$$

$$\text{To minimize } f(\mathbf{x}) = -x_1 x_2 x_3$$

$$\text{Subject to } g_1(\mathbf{x}) \equiv 42 - x_1 \geq 0$$

$$g_2(\mathbf{x}) \equiv 42 - x_2 \geq 0$$

$$g_3(\mathbf{x}) \equiv 42 - x_3 \geq 0$$

$$g_4(\mathbf{x}) \equiv x_1 + 2x_2 + 2x_3 \leq 500$$

$$g_5(\mathbf{x}) \equiv 72 - (x_1 + 2x_2 + 2x_3) \geq 0$$

$$0 \leq x_i \leq 100 \quad i = 1, 2, 3$$

Solution

Step 1: We will rewrite the problem as follows normalizing the constraints.

$$\text{Find } \{x_1, x_2, x_3\}$$

$$\text{To minimize } f(\mathbf{x}) = -x_1 x_2 x_3$$

$$\text{Subject to } g_1(\mathbf{x}) \equiv 1 - x_1/42 \geq 0$$

$$g_2(\mathbf{x}) \equiv 1 - x_2/42 \geq 0$$

$$g_3(\mathbf{x}) \equiv 1 - x_3/42 \geq 0$$

$$g_4(\mathbf{x}) \equiv \frac{x_1 + 2x_2 + 2x_3}{500} \leq 1$$

$$g_5(\mathbf{x}) \equiv 1 - \frac{(x_1 + 2x_2 + 2x_3)}{72} \geq 0$$

$$0 \leq x_i \leq 100 \quad i = 1, 2, 3$$

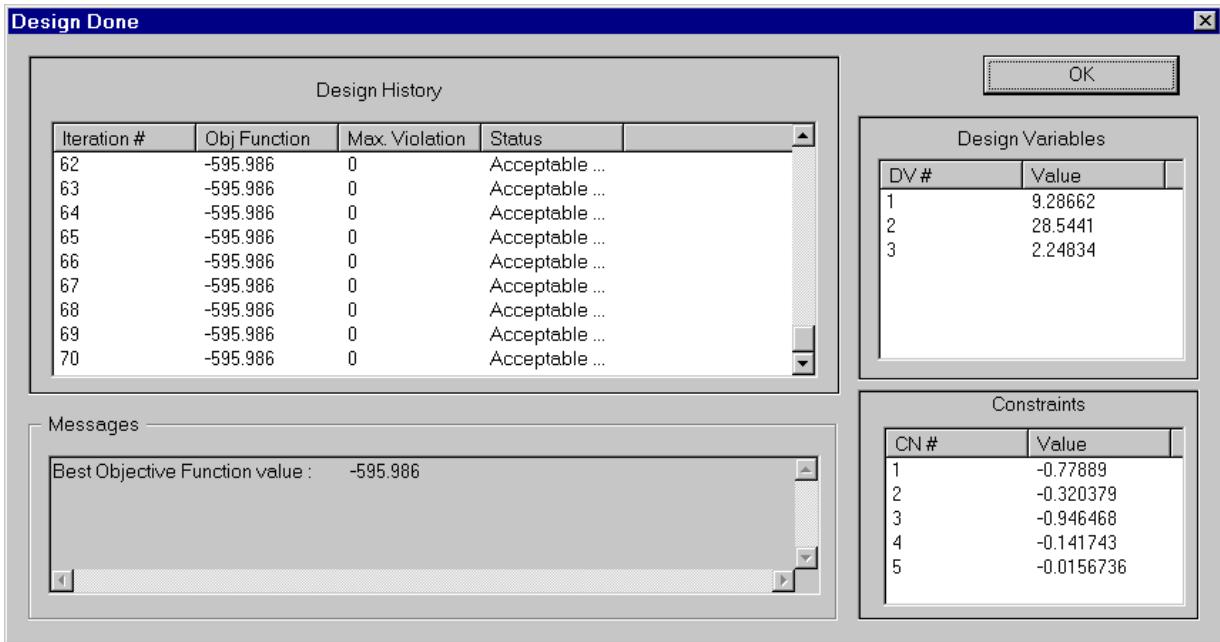
Step 2: Based on the problem formulation, the following variables and functions are necessary.

Variable Name	Remarks
x1	Design variable x1
x2	Design variable x2
x3	Design variable x3

Function Expression	Remarks
$-x_1 \cdot x_2 \cdot x_3$	Objective function
$1-x_1/42$	Constraint g1
$1-x_2/42$	Constraint g2
$1-x_3/42$	Constraint g3
$(x_1+2 \cdot x_2+2 \cdot x_3)/500$	Constraint g4
$1-(x_1+2 \cdot x_2+2 \cdot x_3)/72$	Constraint g5

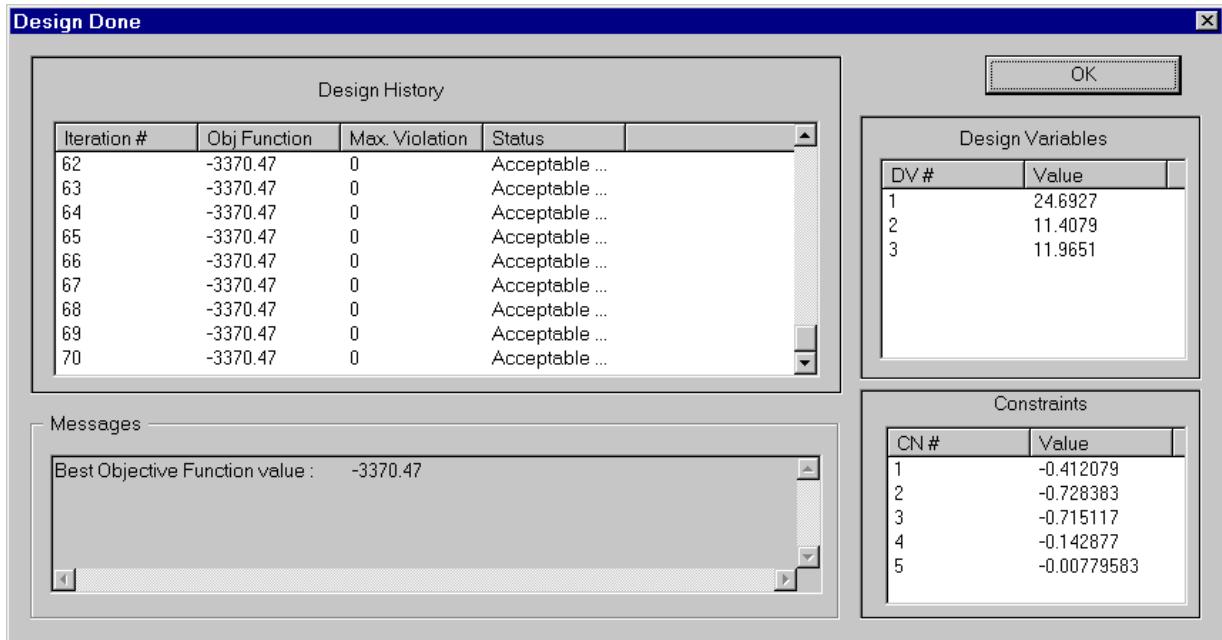
Design Variable	Lower Bound	Upper Bound	Precision
x1	0	100	Auto
x2	0	100	Auto
x3	0	100	Auto

Step 3: The result of executing the GA option in the UNDO[©] program is shown below after 70 generations.



The obtained solution is $\mathbf{x} = \{9.29, 28.5, 2.25\}$. To see whether we can obtain a better solution, we will reduce the upper bound of all the design variables to 30.

Step 4: With the new upper bound, the result of executing the GA option in the UNDO[®] program is shown below after another 70 generations.



The obtained solution is $\mathbf{x} = \{24.7, 11.4, 12\}$ with $f(\mathbf{x}) = -3370$. The optimal solution is $\mathbf{x} = \{24, 12, 12\}$ with $f(\mathbf{x}) = -3456$. The 5th constraint (g_5) controls the design, i.e. is active at the optimum.

Example 17.5-2 Column Design

Fig. E17.5-2 shows a column with a rectangular cross-section that must support an axial force of 100 kN. The column must not fail due to axial stress as well as Euler Buckling (in the X-Y plane). The allowable axial stress is 20 kN/cm^2 . The modulus of elasticity of the material is 1 GPa. The primary objective is to minimize the material volume.

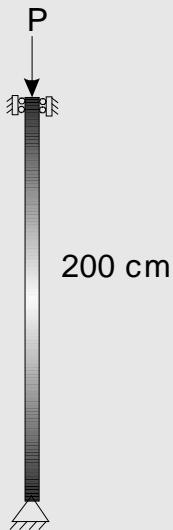


Fig. E17.5-2 Column design

Solution

Step 1: The design problem can be formulated as follows. Converting all quantities to cm and kN ,

(i) the volume of material can be expressed as $200bh \text{ cm}^3$,

(ii) the axial stress in the member is $\frac{100}{bh} \leq 20 \text{ kN/cm}^2$, and

(iii) the Euler buckling requirement can be stated as $P_{cr} = \frac{\pi^2 EI}{L^2} > 100 \text{ kN}$, or

$$1 - \frac{\pi^2 bh^3}{480,000} \leq 0$$

Hence,

$$\text{Find } \mathbf{x} = \{b, h\}$$

$$\text{to minimize } f(\mathbf{x}) = 200bh$$

$$\text{subject to } g_1(\mathbf{x}) = \frac{5}{bh} - 1 \leq 0$$

$$g_2(\mathbf{x}) = 1 - 2.056(10^{-5})bh^3 \leq 0$$

$$g_3(\mathbf{x}) = \frac{b}{h} - 1 \leq 0$$

$$b, h \geq 0$$

Note that all constraints are normalized.

Step 2: Based on the problem formulation, the following variables and functions are necessary.

Variable Name	Remarks
b	Cross-section width
h	Cross-section height

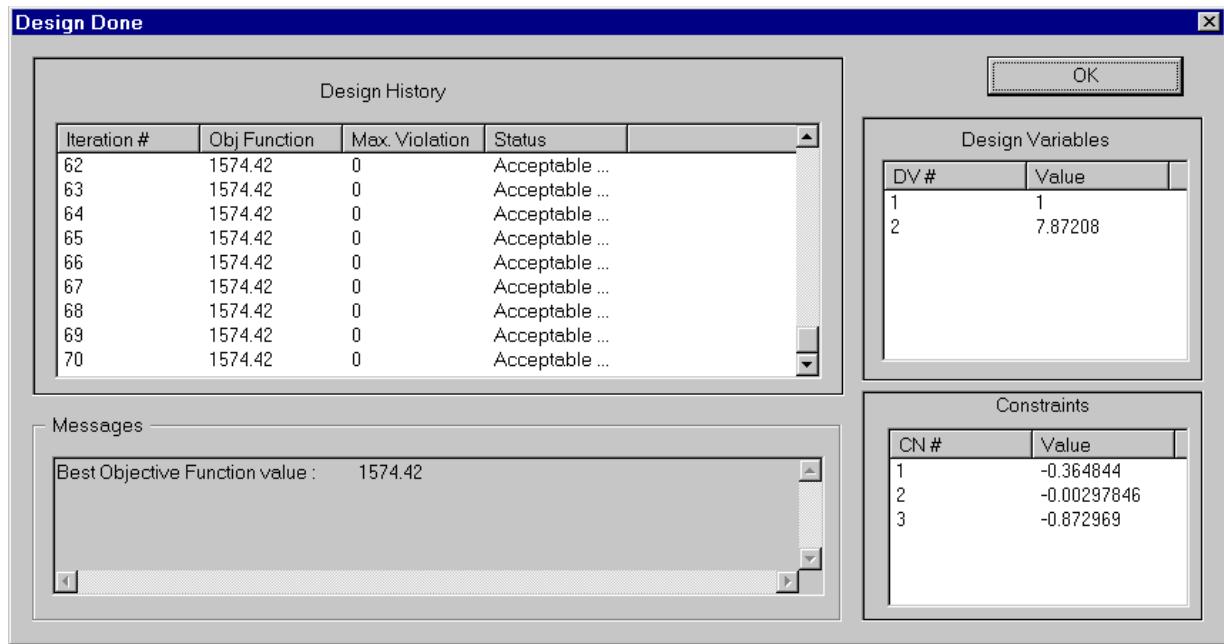
Function Expression	Remarks
200*b*h	Objective function : column volume
5/(b*h)-1	Axial stress constraint
1-2.056e-5*b*h**3	Euler buckling constraint
b/h-1	Cross-section shape

Design Variable	Lower Bound	Upper Bound	Precision
b	1	20	Auto
h	1	20	Auto

The choice of lower and upper bounds should be based on some knowledge of the problem. In this example, the precision is the smallest (or, finest) precision that the program will allow. One should *also* ask the question – When the column is fabricated or constructed, what is the precision (or, tolerance) with which it will be made? The basic strategy is to solve the problem in stages. If at the end of first stage, a refined solution is needed, then you can increase the lower bound, or decrease the upper bound or both. The net effect is that you can then reduce the precision value and (hopefully) obtain a better solution.

Step 3: The result of executing the GA option in the UNDO[©] program is shown below after 70 generations.

The obtained solution is $b = 1\text{ cm}$ and $h = 7.9\text{ cm}$ and $f(\mathbf{x}) = 1574\text{ cm}^3$. This solution is very close to the optimal solution.



The 2nd constraint (g_2) controls the design, i.e. it is active at the optimum.

Example 17.5-3 Beam Design

Fig. E17.5-3 shows a simply-supported beam. The beam is made of Douglas fir (modulus of elasticity = $1,800 \text{ ksi}$, mass density = $1.0 \text{ slugs} / \text{ft}^3$). Find the width b and height h so that the maximum normal stress is less than 2 ksi and the shear stress is less than 0.1 ksi so that the resulting beam is the lightest beam. The height of the beam should not exceed twice the width. Neglect self-weight of the beam.

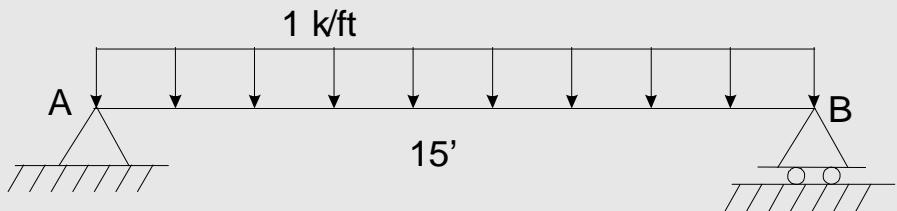


Fig. E17.5-3

Solution

Step 1: Using lb, in as the units, the design problem can be formulated as follows.

$$\text{Find } \{b, h\}$$

$$\text{To minimize } f(b, h) = 180bh$$

$$g_1(\mathbf{x}) = \frac{1.0085(10^3)}{bh^2} - 1 \leq 0$$

$$g_2(\mathbf{x}) = \frac{112.05}{bh} - 1 \leq 0$$

$$g_3(\mathbf{x}) = 1 - \frac{2b}{h} \leq 0$$

$$7'' < h, b < 15''$$

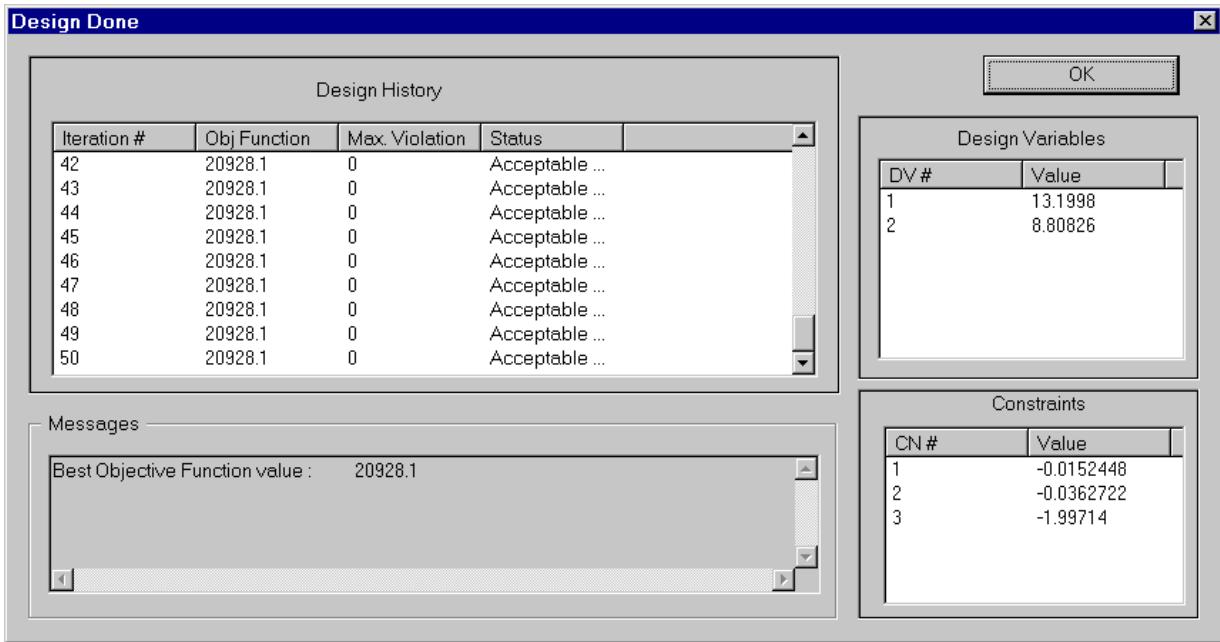
Step 2: Based on the problem formulation, the following variables and functions are necessary.

Variable Name	Remarks
b	Cross-section width
h	Cross-section height

Function Expression	Remarks
$180*b*h$	Objective function : column volume
$1008.5/(b*h^2)-1$	Normal stress constraint
$112.05/(b*h)-1$	Shear stress constraint
$1-(2*b/h)$	Cross-section shape

Design Variable	Lower Bound	Upper Bound	Precision
b	7	15	Auto
h	7	15	Auto

Step 3: The result of executing the GA option in the UNDO[©] program is shown below after 50 generations.



The obtained solution is $b = 13.2"$ and $h = 8.8"$ and $f(\mathbf{x}) = 20928 \text{ in}^3$. This solution is very close to the optimal solution of $b = 12.45"$ and $h = 9.0"$ and $f(\mathbf{x}) = 20169 \text{ in}^3$. The 1st constraint (g_1) and the 2nd constraint (g_2) control the design.

Example 17.5-4 Truss Member Design

Fig. E17.5-4 shows a two-bar truss. Each member is made of a solid square cross-section. The modulus of elasticity, E , is $30(10^6)$ psi. The allowable normal stress is 10,000 psi. The axial stress in the members should be less than the allowable normal stress and the members should satisfy the Euler buckling requirement. The primary objective is to minimize the material volume.

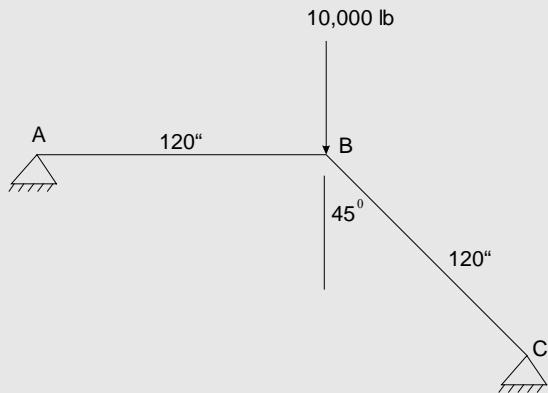


Fig. E17.5-4 Two-Bar Truss design

Solution

Step 1: Using lb and in as the problem units, the optimal design problem can be stated as follows.

$$\begin{aligned}
 \text{Find} \quad & \mathbf{x} = \{a_{AB}, a_{BC}\} \\
 \text{to minimize} \quad & f(\mathbf{x}) = V = 120(a_{AB}^2 + a_{BC}^2) \\
 \text{subject to} \quad & g_1(\mathbf{x}) = \frac{P_{AB}}{10000 a_{AB}^2} - 1 \leq 0 \\
 & g_2(\mathbf{x}) = \frac{P_{BC}}{10000 a_{BC}^2} - 1 \leq 0 \\
 & g_3(\mathbf{x}) = 1 - \frac{(P_{cr})_{AB}}{P_{AB}} \leq 0 \\
 & g_4(\mathbf{x}) = 1 - \frac{(P_{cr})_{BC}}{P_{BC}} \leq 0
 \end{aligned}$$

$$a_{AB}, a_{BC} \geq 0$$

where a_{AB}, a_{BC} cross-section sides for members AB and BC

P_{AB}, P_{BC} magnitude of the axial forces in members AB and BC

$$P_{cr} = \frac{\pi^2 EI}{L^2}$$

represents the Euler buckling capacity of the member

in compression

Using method of joints, $P_{AB} = 10,000$ lb and $P_{BC} = 14,150$ lb with both the members in compression.

Step 2: Based on the problem formulation, the following variables and functions are necessary.

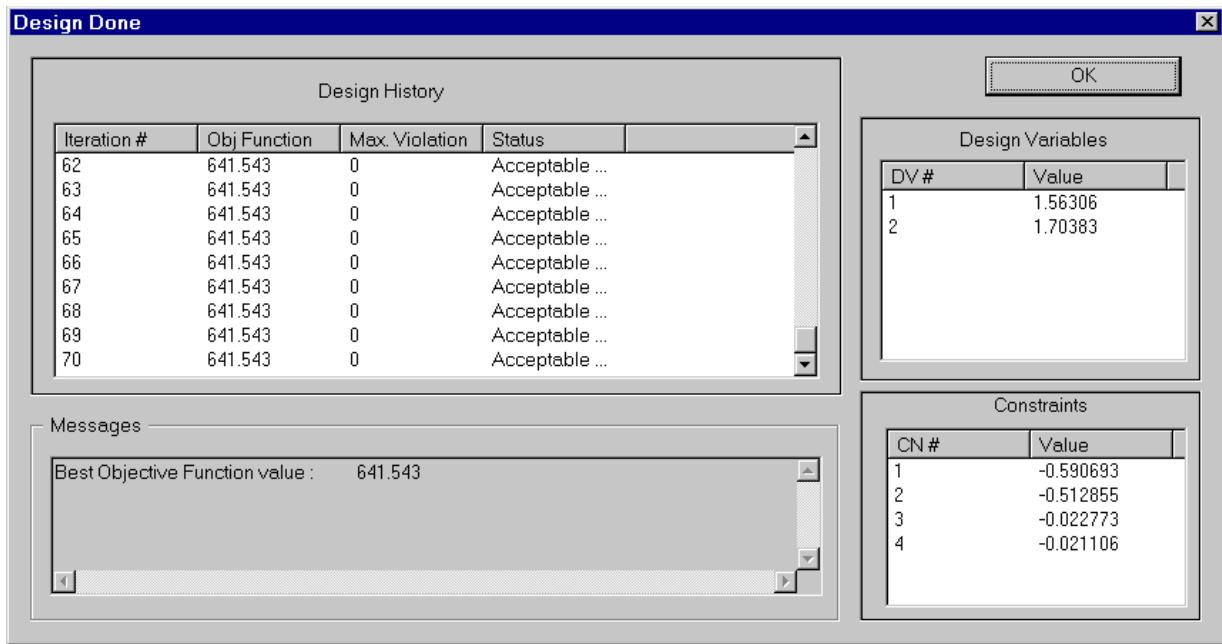
Variable Name	Type	Expression	Remarks
a1	Simple		Cross-section side for member AB
a2	Simple		Cross-section side for member BC
E	Derived	30e6	Modulus of Elasticity
pi	Derived	3.1415926	Pi
I1	Derived	a1**4/12	Moment of inertia (member AB)
I2	Derived	a2**4/12	Moment of inertia (member BC)
stress1	Derived	10000/a1**2	Stress in member AB
stress2	Derived	14150/a2**2	Stress in member BC

Function Expression	Remarks
120*(a1**2+a2**2)	Objective function : Truss volume
stress1/10000-1	Axial stress constraint : member AB
Stress2/10000-1	Axial stress constraint : member BC
1-(pi**2*E*I1)/(120**2*10000)	Euler Buckling : member AB
1-(pi**2*E*I2)/(120**2*14150)	Euler Buckling : member BC

Design Variable	Lower Bound	Upper Bound	Precision
a1	1	2	Auto
a2	1	2	Auto

This example illustrates how to use simple and derived variables to reduce the amount of hand-calculations necessary to formulate the design problem. We could have used more derived variables than shown above.

Step 3: The result of executing the GA option in the UNDO[©] program is shown below after 70 generations.



The obtained solution is $a_1 = 1.56"$ and $a_2 = 1.70"$ and $f(\mathbf{x}) = 641.5 \text{ in}^3$. This solution is very close to the optimal solution. The 3rd constraint (g_3) and the 4th constraint (g_4) control the design.

EXERCISES

Appetizers

Solve the following problems using pencil and paper.

Problem 17.5-1

Fig. P17.5-1 shows a determinate planar beam. The cross-section is rectangular (height h and width w). The allowable normal stress is 12 MPa and the shear stress is 5 MPa. Design the minimum volume beam so that the height is not more than twice the width.

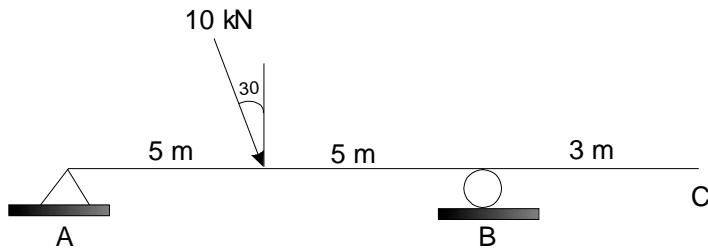


Fig. P17.5-1

Problem 17.5-2

How would you formulate and solve the previous problem if the weight density of the beam material was given as 6000 N/m^2 ?

Main Course

Problem 17.5-3

A steel pipe is moved to place by a crane using the system shown in Fig. P17.5-3. The inner pipe diameter is 40" and the wall thickness is 0.375". The length of the pipe is 24 ft. The maximum axial load capacity of the cable is 200 000 lb. Find the distance d between the lifting points to minimize the maximum bending stress in the pipe.

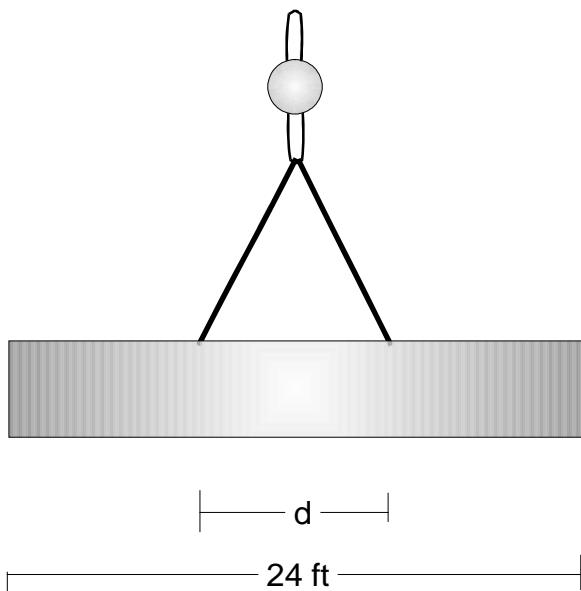


Fig. P17.5-3

Problem 17.5-4

Fig. P17.5-5 shows a planar two-bar truss. Support C is located directly above A. The allowable stress in member AB is 100 MPa and in member BC is 200 MPa. Find the cross-sectional areas of members AB and BC, and the distance x to design the minimum volume truss.

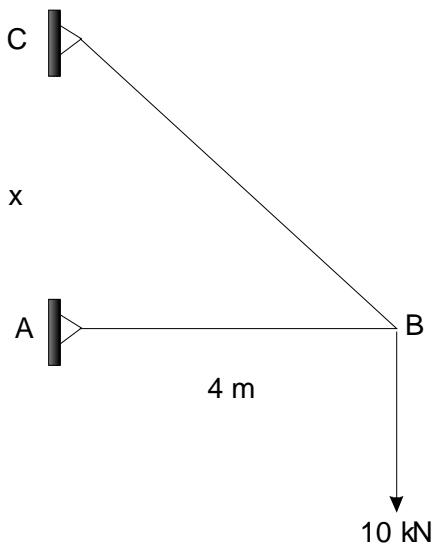


Fig. P17.5-4

C++ Concepts

Problem 17.5-5

Develop a class **CSGA** that implements a simple GA, using object-oriented concepts.

Solve the following problems using computer software such as UNDO[©].

Problem 17.5-6

It is required to design a support bracket as shown in Fig. P17.5-5. Member ADC is W16x31. Member BD has a circular hollow cross-section. Supports A and B are pin supports and connection at D is a pin connection. Design the lightest steel member BD so that the normal stress in the member is less than 10 ksi and Euler buckling is prevented with a safety factor of 2. The wall thickness of the pipe cannot be less than 15% of the inner radius. Also find the optimal values for x and d .

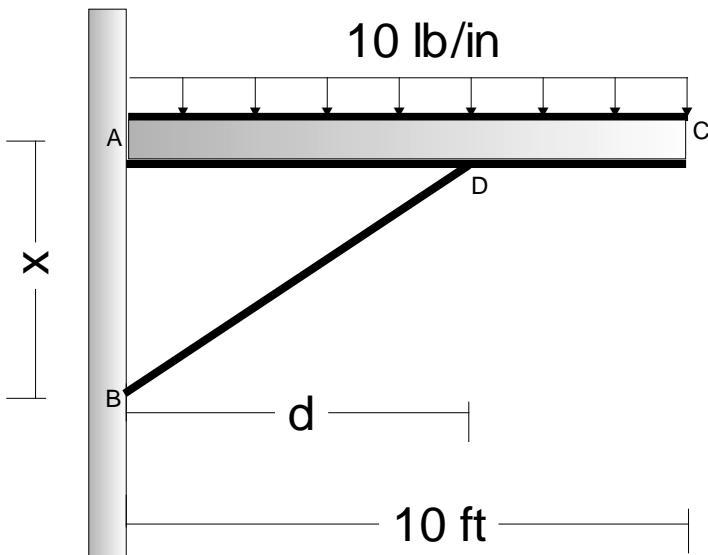


Fig. P17.5-5

Chapter

18

Computer Graphics

“Ambition is a lust that is never quenched, but grows more inflamed and madder by enjoyment.”

Thomas Otway

“Build a better mousetrap and the world will beat a path to your door.” Ralph Waldo

Emerson

“It is not the greatness of a man's means that makes him independent, so much as the smallness of his wants”. William Cobbett

In this chapter we will study some of the basics of computer graphics.

Objectives

- To understand and practice the concepts associated with computer graphics.

18.1 Introduction

18.2 Simple Operations

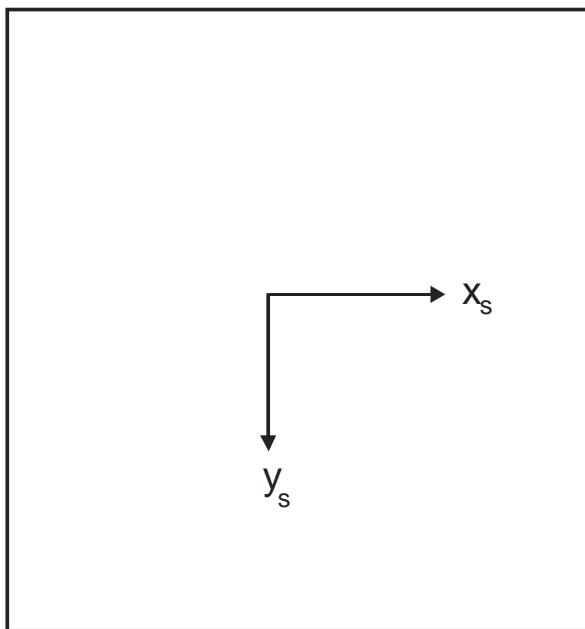


Fig. 18.2.1 Screen coordinate system (note left-handed system)

18.3 An X-Y Graphing Program

18.4 Transformations and Projections

Translation

Translation of a point (x,y,z) through (a,b,c) can be simply achieved by computing the new coordinates as

$$\begin{Bmatrix} x \\ y \\ z \end{Bmatrix}_{new} = \begin{Bmatrix} x \\ y \\ z \end{Bmatrix} + \begin{Bmatrix} a \\ b \\ c \end{Bmatrix}$$

Rotation about Coordinate Axes

The transformation matrices for rotation about the three coordinate axes are given below.

Rotation about **x**

$$\mathbf{T}_{3\times 3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & \sin \theta \\ 0 & -\sin \theta & \cos \theta \end{bmatrix}$$

Rotation about **y**

$$\mathbf{T}_{3\times 3} = \begin{bmatrix} \cos \theta & 0 & -\sin \theta \\ 0 & 1 & 0 \\ \sin \theta & 0 & \cos \theta \end{bmatrix}$$

Rotation about **z**

$$\mathbf{T}_{3\times 3} = \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Example 1: Consider a triangle lying on the x-y plane with coordinates as (3,-1,0), (4,1,0) and (2,1,0). Consider a counterclockwise (or positive) rotation of 90° about the z-axis. The transformation matrix is then

$$\mathbf{T}_{3\times 3} = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Hence, the new coordinates of the three points can be computed as $\mathbf{P}_{3\times 3}\mathbf{T}_{3\times 3}$ where $\mathbf{P}_{3\times 3}$ is the matrix with each row containing the (x,y,z) coordinates of the corresponding point. Hence

$$\begin{bmatrix} 3 & -1 & 0 \\ 4 & 1 & 0 \\ 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 0 \\ -1 & 4 & 0 \\ -1 & 2 & 0 \end{bmatrix}$$

Example 2: Consider a triangle lying on the x-z plane with coordinates as (0,0,0), (0,0,3) and (5,0,0). Consider a counterclockwise (or positive) rotation of 90° about the y-axis. The transformation matrix is then

$$\mathbf{T}_{3 \times 3} = \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Hence, the new coordinates of the three points can be computed as $\mathbf{P}_{3 \times 3} \mathbf{T}_{3 \times 3}$ where $\mathbf{P}_{3 \times 3}$ is the matrix with each row containing the (x,y,z) coordinates of the corresponding point. Hence

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 3 \\ 5 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 3 & 0 & 0 \\ 0 & 0 & -5 \end{bmatrix}$$

18.5 Three-Dimensional Graphics

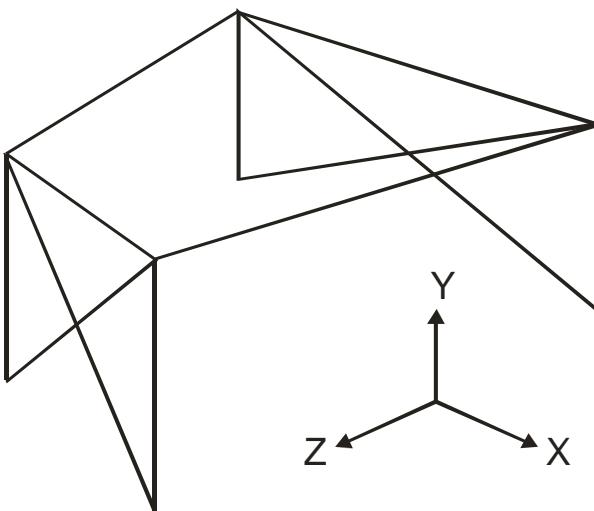


Fig. 18.5.1 Physical or model coordinate system

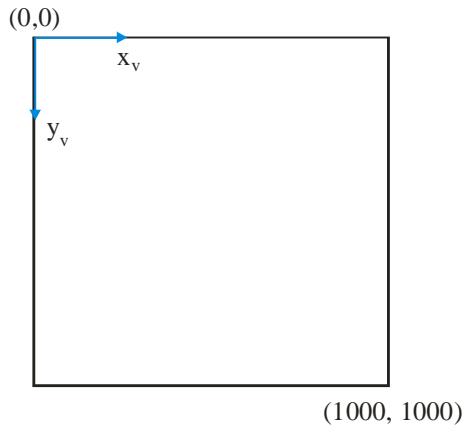


Fig. 18.5.2 Virtual coordinate system

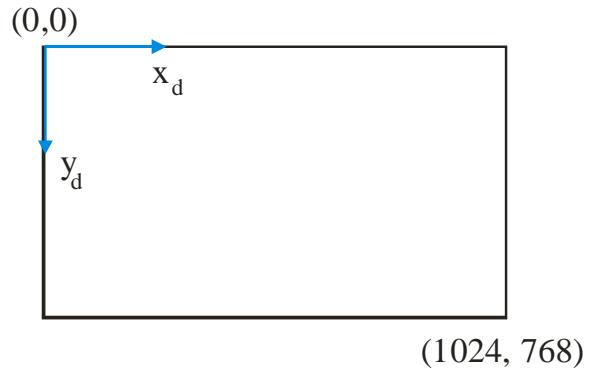


Fig. 18.5.3 An example device coordinate system

Algorithm

Step 1 Compute model limits in physical or model coordinates

Loop thro' all nodes to compute $(X_{\min}, X_{\max}), (Y_{\min}, Y_{\max}), (Z_{\min}, Z_{\max})$.

Compute center of model as $(X_{mid}, Y_{mid}, Z_{mid})$. For example, $X_{mid} = \frac{X_{\min} + X_{\max}}{2}$.

Set $\mathbf{T}_{3 \times 3} = \mathbf{I}_{3 \times 3}$ as the initial transformation matrix.

Step 2 Compute scaling factor

Find coordinate translation to center the viewing box in the virtual coordinate

$$\text{system using the formula } \begin{Bmatrix} X_i \\ Y_i \\ Z_i \end{Bmatrix} = \mathbf{T}_{3 \times 3} \left[\begin{Bmatrix} X \\ Y \\ Z \end{Bmatrix} - \begin{Bmatrix} X_{mid} \\ Y_{mid} \\ Z_{mid} \end{Bmatrix} \right]_{3 \times 1}$$

Apply the above formula for the 8 corners of the viewing box and compute

$(X_{\min}^v, X_{\max}^v), (Y_{\min}^v, Y_{\max}^v), (Z_{\min}^v, Z_{\max}^v)$ using those 8 transformed coordinates.

Compute the coordinates of the center of the model as

$$\begin{Bmatrix} X_b \\ Y_b \\ Z_b \end{Bmatrix} = \frac{1}{2} \left[\begin{Bmatrix} X_{\min}^v + X_{\max}^v \\ Y_{\min}^v + Y_{\max}^v \\ Z_{\min}^v + Z_{\max}^v \end{Bmatrix} \right]_{3 \times 1}$$

Compute scaling factor $s = \min \left(\frac{A-a}{X_{\max}^v - X_{\min}^v}, \frac{A-a}{Y_{\max}^v - Y_{\min}^v} \right)$

Step 3 Draw the truss

Loop thro' all elements.

For the start node, compute the virtual coordinates.

$$\begin{Bmatrix} X_i \\ Y_i \\ Z_i \end{Bmatrix} = \mathbf{T}_{3 \times 3} \left[\begin{Bmatrix} X \\ Y \\ Z \end{Bmatrix} - \begin{Bmatrix} X_{mid} \\ Y_{mid} \\ Z_{mid} \end{Bmatrix} \right]_{3 \times 1}$$

$$\begin{Bmatrix} x_v \\ y_v \end{Bmatrix} = s \left[\begin{Bmatrix} X_i \\ Y_i \end{Bmatrix} - \begin{Bmatrix} X_b \\ Y_b \end{Bmatrix} \right] + \begin{Bmatrix} A/2 \\ A/2 \end{Bmatrix}$$

Finally compute the device coordinates using one of the following formula

$$\begin{Bmatrix} x_d \\ y_d \end{Bmatrix} = \alpha \begin{Bmatrix} x_v \\ y_v \end{Bmatrix} + \begin{Bmatrix} 0.5 \\ 0.5 \end{Bmatrix}$$

$$\begin{Bmatrix} x_d \\ y_d \end{Bmatrix} = \begin{Bmatrix} \alpha_x x_v + 0.5 \\ \alpha_y y_v + 0.5 \end{Bmatrix}$$

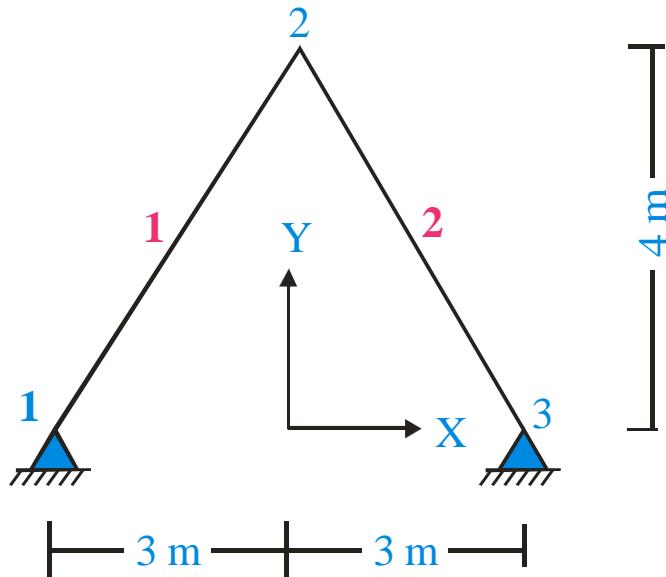
where $\alpha_x = x_d^{range}/A$, $\alpha_y = y_d^{range}/A$ and $\alpha = \min(\alpha_x, \alpha_y)$.

Move to (x_d, y_d) .

Repeat these computations for the end node and draw to (x_d, y_d) .

Example 1 (Planar Truss)

Consider the truss shown in Example 6.2.1. Draw the initial view of the truss assuming that $A = 1000$, $a = 100$ and $(x_d^{range}, y_d^{range}) = (1024, 768)$.



Solution

Step 1: Here are the results.

$$(X_{\min}, X_{\max}) = (-3, 3), (Y_{\min}, Y_{\max}) = (0, 4) \text{ and } (Z_{\min}, Z_{\max}) = (0, 0).$$

$$(X_{mid}, Y_{mid}, Z_{mid}) = (0, 2, 0).$$

Step 2: The corners of the viewing box have the following coordinates before and after translating to the virtual coordinate system.

Corner	Physical Coordinates	Virtual Coordinates
1	(-3, 0, 0)	(-3, -2, 0)
2	(-3, 0, 0)	(-3, -2, 0)
3	(-3, 4, 0)	(-3, 2, 0)
4	(-3, 4, 0)	(-3, 2, 0)
5	(3, 0, 0)	(3, -2, 0)
6	(3, 0, 0)	(3, -2, 0)
7	(3, 4, 0)	(3, 2, 0)
8	(3, 4, 0)	(3, 2, 0)

Hence, $(X_{\min}^v, X_{\max}^v) = (-3, 3)$, $(Y_{\min}^v, Y_{\max}^v) = (-2, 2)$ and $(Z_{\min}^v, Z_{\max}^v) = (0, 0)$. Also, $(X_b, Y_b, Z_b) = (0, 0, 0)$. The scaling factor can then be computed as

$$s = \min\left(\frac{1000-100}{3+3}, \frac{1000-100}{2+2}\right) = 150$$

Step 3: For each element, here are the computed numbers.

Element 1, Start Node

$$\begin{cases} X_i \\ Y_i \\ Z_i \end{cases} = \mathbf{T}_{3 \times 3} \begin{bmatrix} \begin{cases} -3 \\ 0 \\ 0 \end{cases} & \begin{cases} 0 \\ 2 \\ 0 \end{cases} \end{bmatrix}_{3 \times 1} = \begin{cases} -3 \\ -2 \\ 0 \end{cases}$$

$$\begin{cases} x_v \\ y_v \end{cases} = 150 \begin{bmatrix} \begin{cases} -3 \\ -2 \end{cases} & \begin{cases} 0 \\ 0 \end{cases} \end{bmatrix} + \begin{cases} 500 \\ 500 \end{cases} = \begin{cases} 50 \\ 200 \end{cases}$$

Isotropic Mapping

$$\begin{cases} x_d \\ y_d \end{cases} = \alpha \begin{cases} x_v \\ y_v \end{cases} + \begin{cases} 0.5 \\ 0.5 \end{cases} = 0.768 \begin{cases} 50 \\ 200 \end{cases} + \begin{cases} 0.5 \\ 0.5 \end{cases} = \begin{cases} 39 \\ 154 \end{cases}$$

Anisotropic Mapping

$$\begin{cases} x_d \\ y_d \end{cases} = \begin{cases} \alpha_x x_v + 0.5 \\ \alpha_y y_v + 0.5 \end{cases} = \begin{cases} 1.024 \times 50 + 0.5 \\ 0.768 \times 200 + 0.5 \end{cases} = \begin{cases} 52 \\ 154 \end{cases}$$

Element 1, End Node

$$\begin{cases} X_i \\ Y_i \\ Z_i \end{cases} = \mathbf{T}_{3 \times 3} \left[\begin{cases} 0 \\ 4 \\ 0 \end{cases} - \begin{cases} 0 \\ 2 \\ 0 \end{cases} \right]_{3 \times 1} = \begin{cases} 0 \\ 2 \\ 0 \end{cases}$$

$$\begin{cases} x_v \\ y_v \end{cases} = 150 \left[\begin{cases} 0 \\ 2 \\ 0 \end{cases} - \begin{cases} 0 \\ 0 \\ 0 \end{cases} \right] + \begin{cases} 500 \\ 500 \end{cases} = \begin{cases} 500 \\ 800 \end{cases}$$

Isotropic Mapping

$$\begin{cases} x_d \\ y_d \end{cases} = \alpha \begin{cases} x_v \\ y_v \end{cases} + \begin{cases} 0.5 \\ 0.5 \end{cases} = 0.768 \begin{cases} 500 \\ 800 \end{cases} + \begin{cases} 0.5 \\ 0.5 \end{cases} = \begin{cases} 385 \\ 615 \end{cases}$$

Anisotropic Mapping

$$\begin{cases} x_d \\ y_d \end{cases} = \begin{cases} \alpha_x x_v + 0.5 \\ \alpha_y y_v + 0.5 \end{cases} = \begin{cases} 1.024 \times 500 + 0.5 \\ 0.768 \times 800 + 0.5 \end{cases} = \begin{cases} 513 \\ 615 \end{cases}$$

Element 2, Start Node

The results are the same as end node for element 1.

Element 2, End Node

$$\begin{cases} X_i \\ Y_i \\ Z_i \end{cases} = \mathbf{T}_{3 \times 3} \left[\begin{cases} 3 \\ 0 \\ 0 \end{cases} - \begin{cases} 0 \\ 2 \\ 0 \end{cases} \right]_{3 \times 1} = \begin{cases} 3 \\ -2 \\ 0 \end{cases}$$

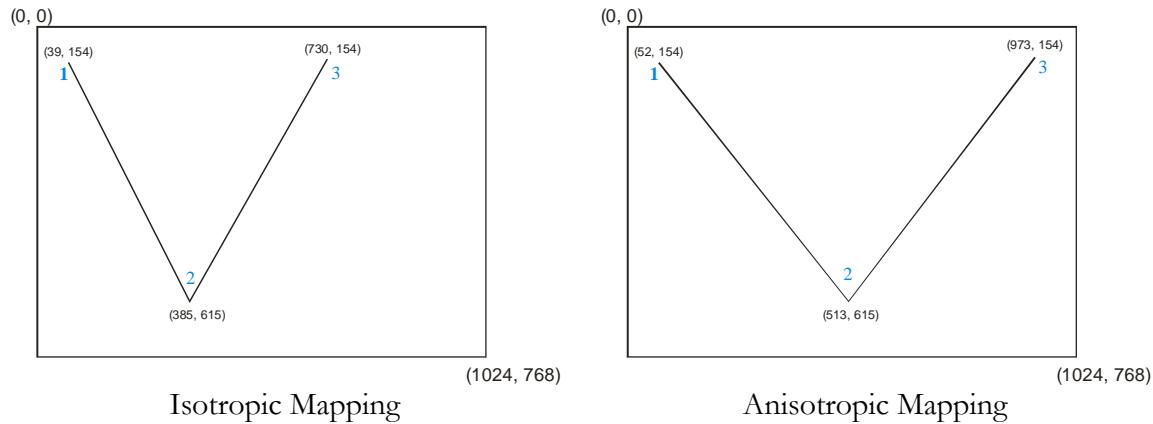
$$\begin{cases} x_v \\ y_v \end{cases} = 150 \left[\begin{cases} 3 \\ -2 \\ 0 \end{cases} - \begin{cases} 0 \\ 0 \\ 0 \end{cases} \right] + \begin{cases} 500 \\ 500 \end{cases} = \begin{cases} 950 \\ 200 \end{cases}$$

Isotropic Mapping

$$\begin{cases} x_d \\ y_d \end{cases} = \alpha \begin{cases} x_v \\ y_v \end{cases} + \begin{cases} 0.5 \\ 0.5 \end{cases} = 0.768 \begin{cases} 950 \\ 200 \end{cases} + \begin{cases} 0.5 \\ 0.5 \end{cases} = \begin{cases} 730 \\ 154 \end{cases}$$

Anisotropic Mapping

$$\begin{cases} x_d \\ y_d \end{cases} = \begin{cases} \alpha_x x_v + 0.5 \\ \alpha_y y_v + 0.5 \end{cases} = \begin{cases} 1.024 \times 950 + 0.5 \\ 0.768 \times 200 + 0.5 \end{cases} = \begin{cases} 973 \\ 154 \end{cases}$$



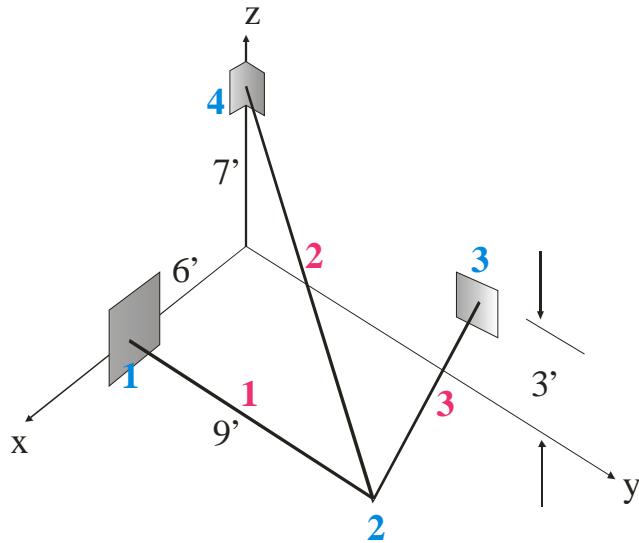
The truss appears to be inverted and it is! Hence we need to change the default transformation matrix as follows.

$$\mathbf{T}_{3 \times 3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Example 2: Draw the truss in Example 1 after rotating 20° counterclockwise about the z-axis.
Solution

Example 3 (Space Truss)

Consider the space truss shown below. Draw the initial view of the truss assuming that $A = 1000$, $a = 100$ and $(x_d^{range}, y_d^{range}) = (1024, 768)$.



18.6 Case Study: 3D Wireframe Viewer

Summary

EXERCISES

Appetizers

Main Course

C++ Concepts



Using Microsoft Visual Studio

'When you get to the fork in the road, take it.' "Yogi Berra

'If you do not know where you are going, you'll wind up somewhere else.' "Yogi Berra

'It's déjà vu all over again.' "Yogi Berra

1.0 Introduction

Microsoft Visual Studio (MSVS) is an Integrated Development Environment (IDE) that programmers use to develop (create, debug and maintain) and launch computer programs written in a number of high-level languages. In this document we will see how two types of computer programs – console applications and MFC applications can be developed using C++. We will also see how to use the debugging features of MS VS.

Console applications are programs where user input is text-based typically via a keyboard and/or external file and output is also text-based typically to the console (computer monitor) and/or external file. MFC (Microsoft Foundation Class) is a collection of classes that is provided with VS 2005 to help developers build graphical-user interface (GUI) programs to work with various flavors of Windows OS.

There are several differences between a console application and an MFC application that the reader must be aware of. Here is a short but important list.

- (1) The usual std::cout and std::cin will not work with an MFC application.
- (2) There is no main program – MFC applications are event-driven.
- (3) MFC provides resources normally associated with Windows applications – toolbar, menu, status bar etc.
- (4) MFC applications use precompiled headers.

2.0 Building C++ Applications

In this section we will first see how to develop a console application followed by the steps for a MFC application.

2.1 Building a Console Application Suitable for Debugging

Step 1: Select a Win32 Console Application

Launch Visual Studio. The program interface should look similar to Fig. 2.1.1.

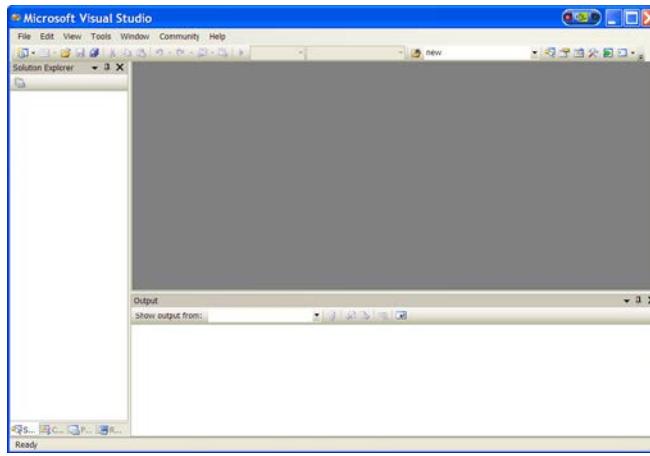


Fig. 2.1.1 Graphical User Interface upon launching MS VS 2005

Click **File**, **New** and then **Project**. Make sure that Visual C++, Win32 and Win32 Console Application are selected as shown in Fig. 2.1.2.

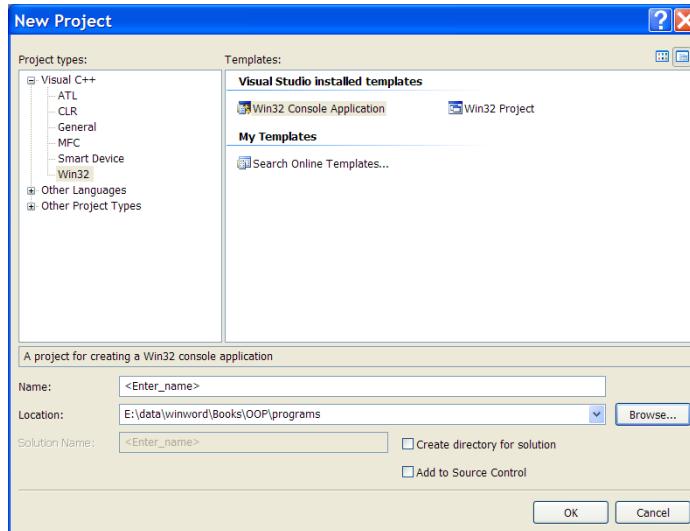


Fig. 2.1.2 Selecting a Win32 Console Application

Type in the **Name** of the project (for example, NumDiff) and make sure the location points to the correct directory. Click the **OK** button. This launches the Win32 Application Wizard (see Fig. 2.1.3).

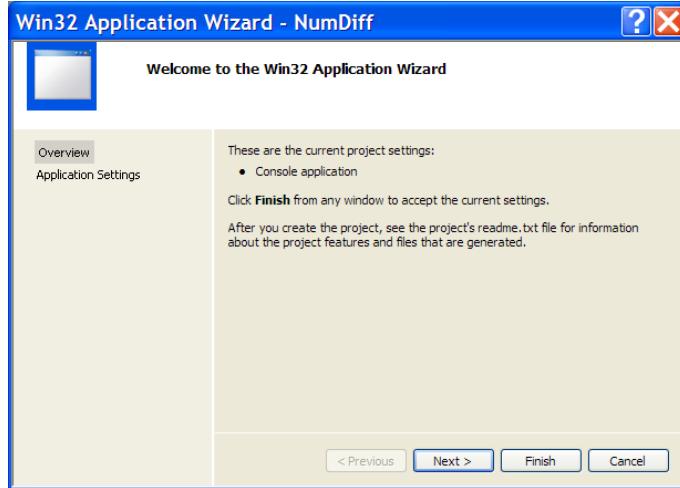


Fig. 2.1.3 Win32 Application Wizard

Step 2: Specify the type of Console Application

Follow these steps.

(a) Click **Application Settings** and make the selections as shown below (Fig. 2.1.4).

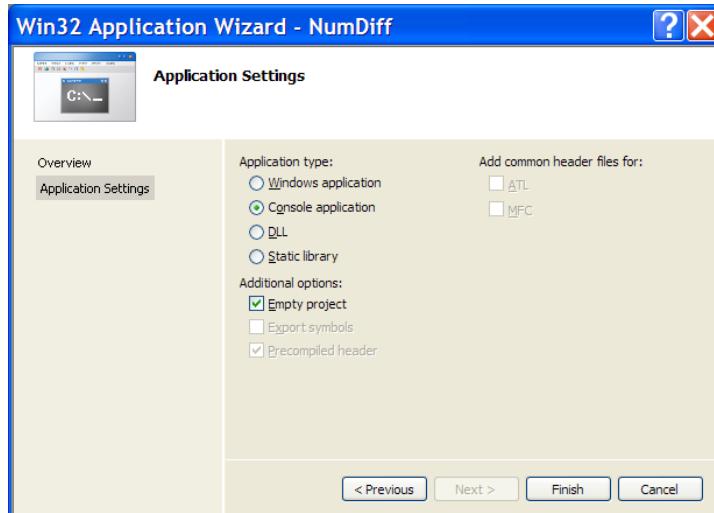


Fig. 2.1.4 Selections for a console application

Note the selections – **Console application** as the Application Type and with the **Empty project** selected for Additional options.

(b) Click the **Finish** button. You are now ready to start creating your application (see Fig. 2.1.5). Note that the word **Debug** appears in the leftmost combo box just below the main window. The other option in this combo box is **Release**. Debugging instructions are inserted in the executable image if the Debug option is selected. The debugging process is discussed in Section 3.

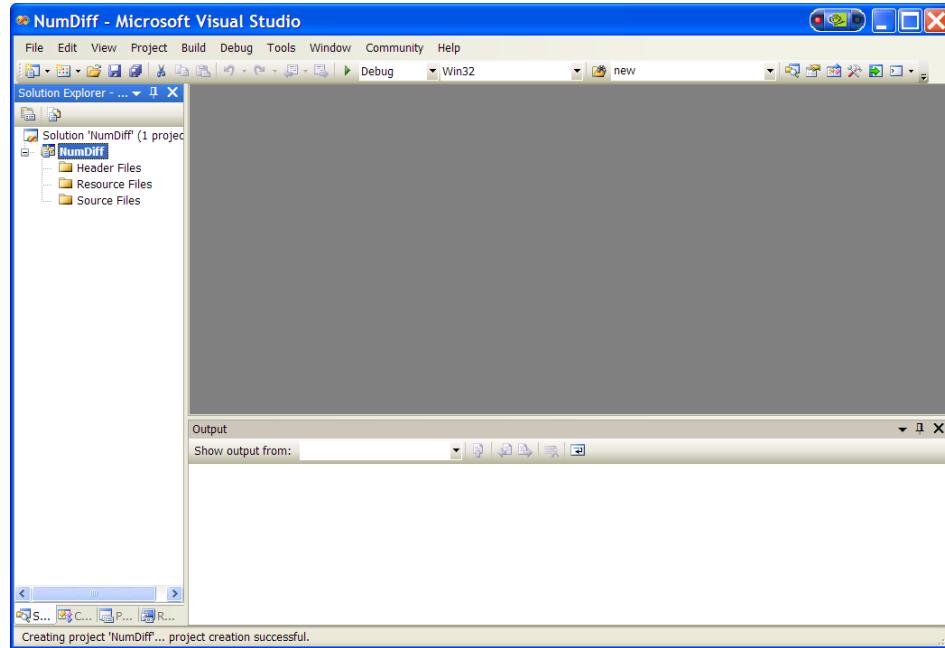


Fig. 2.1.5 VS user interface showing status ready for program development

Step 3: Developing the Source Code

We are now ready to develop the source code one file at a time. In this example, we will show how to create one header file and one source file. Let us create the source file first. Click **Project**, followed by **Add New Item**. Fig. 2.1.6 shows the dialog box that is launched.

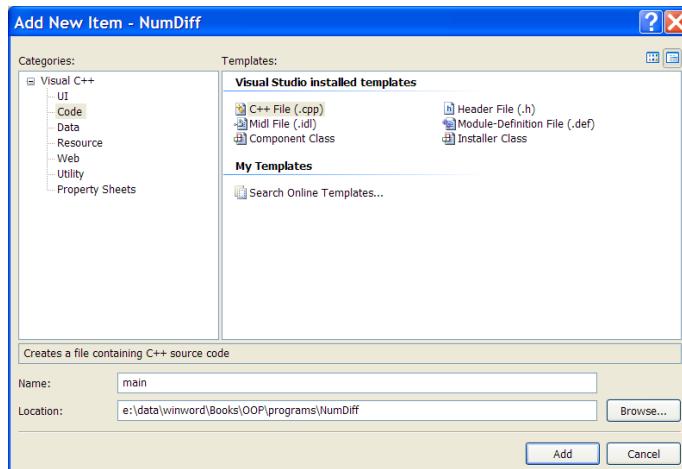


Fig. 2.1.6 Dialog box for creating source code

Note that **Code** and **C++ File (.cpp)** are selected. Enter the **Name** of the file (for example, main). Click the **Add** button. This launches the editor and you are now ready to type in the source code (Fig. 2.1.7). Note that the file name, main.cpp, appears under the Source Files section in the Solution Explorer window. In addition, the mouse cursor appears in the first column and first line of the C++ editor window.

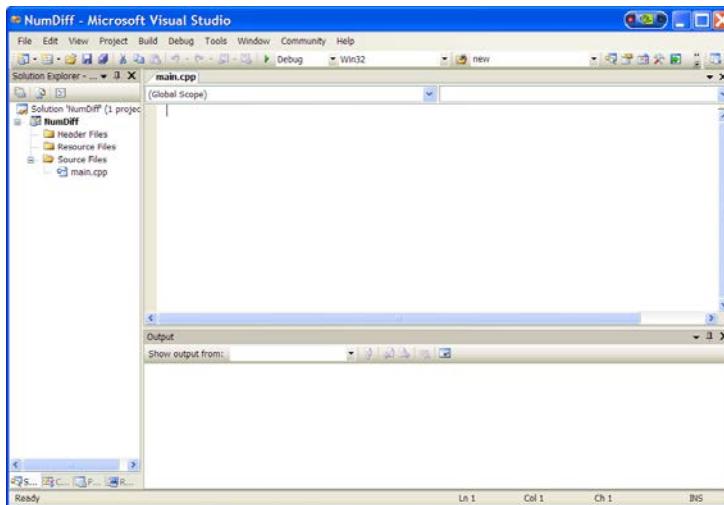


Fig. 2.1.7 Program interface showing C++ editor ready to accept creation of file called main.cpp

Type in the C++ statements as shown in Fig. 2.1.8. Note that we have not saved the file yet (hint: main.cpp* indicates that the file contents have changed and that the file has not been saved).

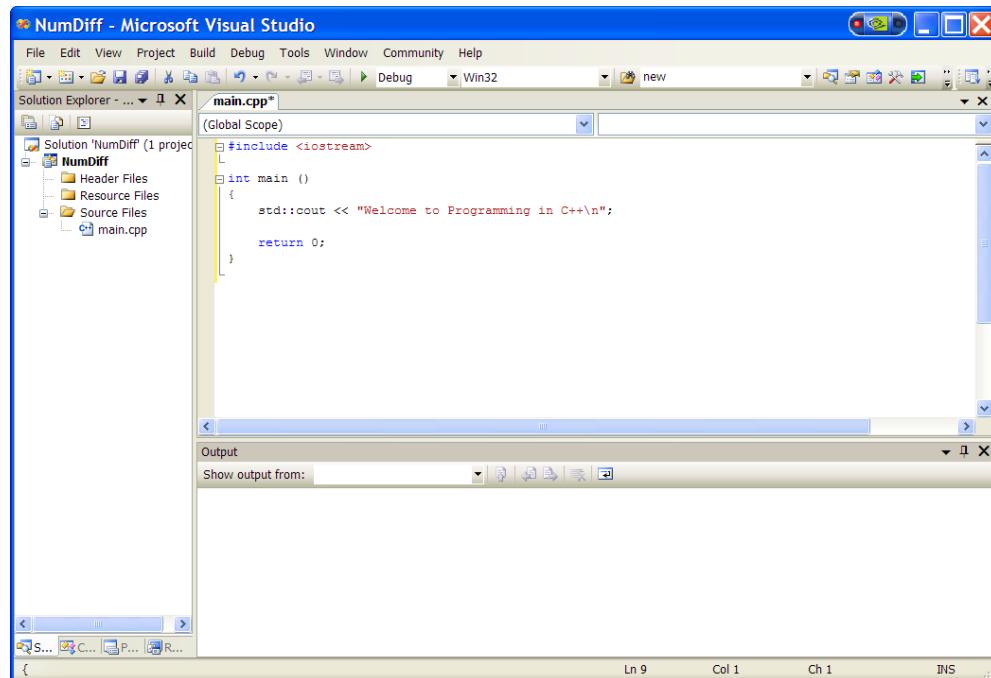


Fig. 2.1.8 A complete Hello World-type of C++ program

Step 4: Compiling, Linking and Executing the program

We are now ready to compile, link and execute the program. Click **Build** and then **Build Solution**. This compiles and links the program. In other words, contents of the cpp file are used by the compiler to create an object file (.obj file) and if successful, the linker uses the object file to create the executable file (.exe file). Fig. 2.1.9 shows success in compiling and linking the program to create the executable numdiff.exe.

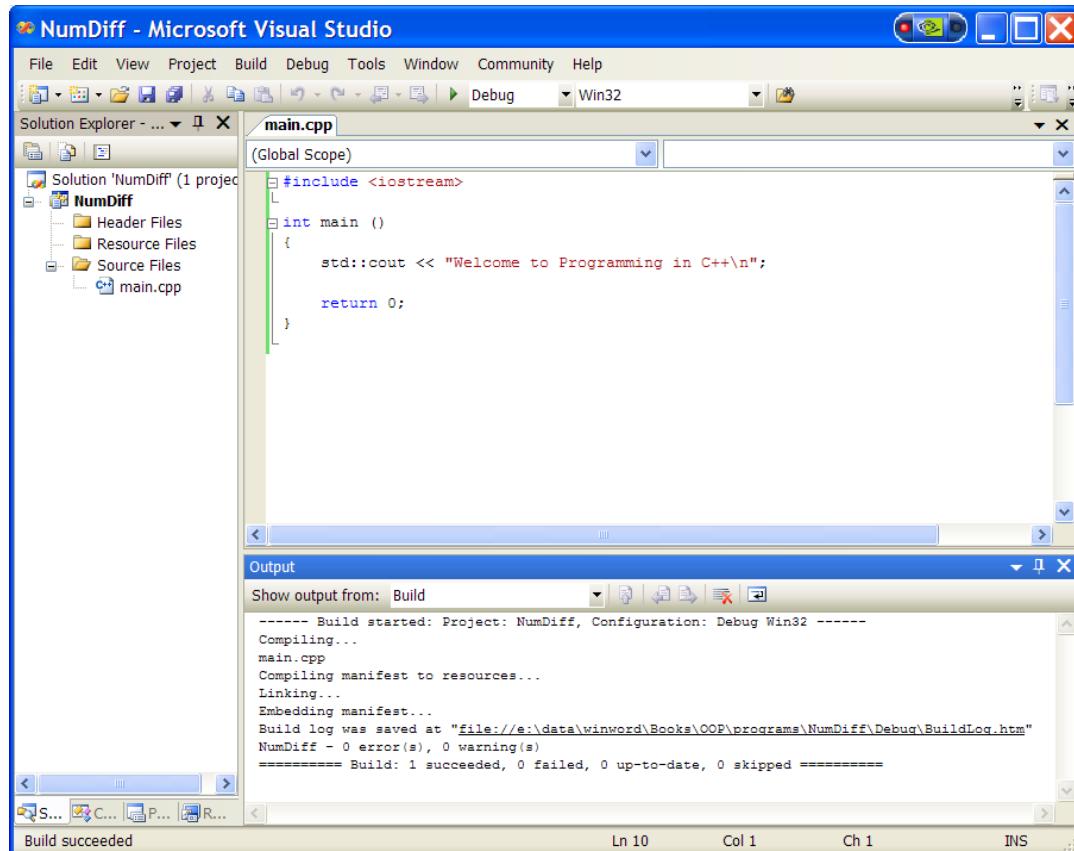


Fig. 2.1.9 A successful compile and link steps

Now we are ready to execute the program. Click **Debug** and then **Start Without debugging**. The executable file is executed and since this is a console application, a DOS-like window appears with the output that we expect to see (Fig. 2.1.10). Press any key to close the console window.

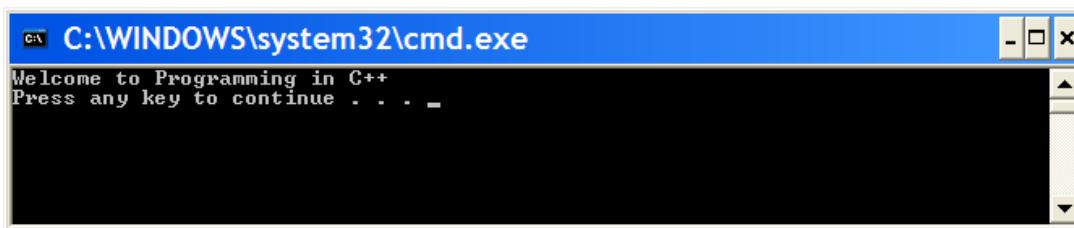


Fig. 2.1.10 Successful execution of the program

Step 5: Modifying the Program and Adding a header file

We will modify the main program by creating a function that would output the welcome message on the console. To edit the main program, double click **main.cpp** in the Solution Explorer. Make the changes as shown in Fig. 2.1.11.

```

NumDiff - Microsoft Visual Studio
File Edit View Project Build Debug Tools Window Community Help
Solution Explorer ... X
Solution 'NumDiff' (1 project)
  NumDiff
    Header Files
    Resource Files
    Source Files
      main.cpp

main.cpp (Global Scope) X
int main ()
{
    ShowMessage ();
    return 0;
}

```

Fig. 2.1.11 Modified main program

Now we are ready to create the function `ShowMessage()` first in a separate file `showmessage.h` and then the actual function in `showmessage.cpp`. Click **Project** and then **Add New Item**. Unlike before (Fig. 2.1.6) select **Header File (.h)**. Type the name of the file as `showmessage`. Finally, type in the statements shown in Fig. 2.1.12.

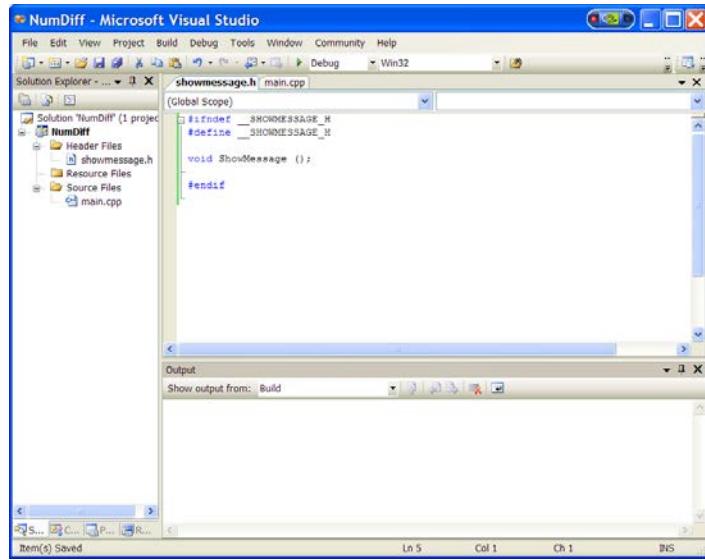


Fig. 2.1.12 showmessage header file

Click **Project** and then **Add New Item**. Select **C++ File (.cpp)**. Type the name of the file as showmessage. Finally, type in the statements shown in Fig. 2.1.13.

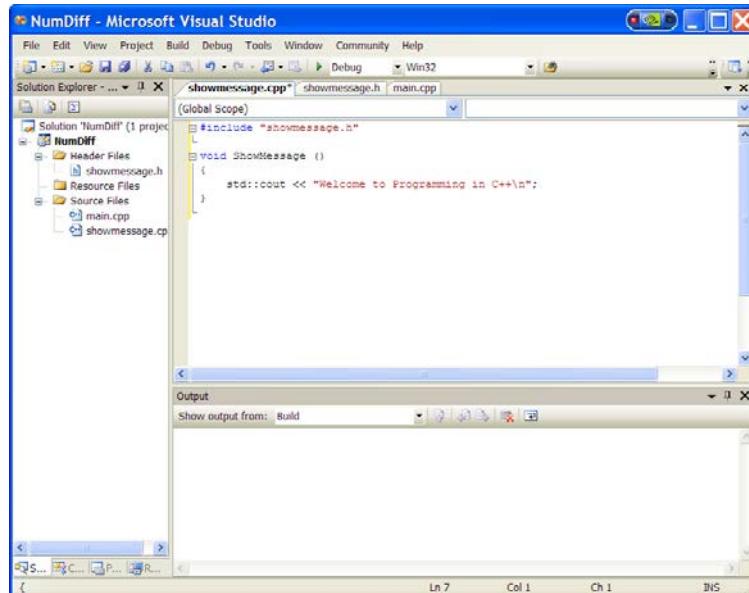


Fig. 2.1.13 ShowMessage() function contained in file showmessage.cpp

Step 6: Compiling, Linking and Executing the new version of the program

Click **Build** and then **Build Solution**. The program does not compile! The error messages appear in the Output window.

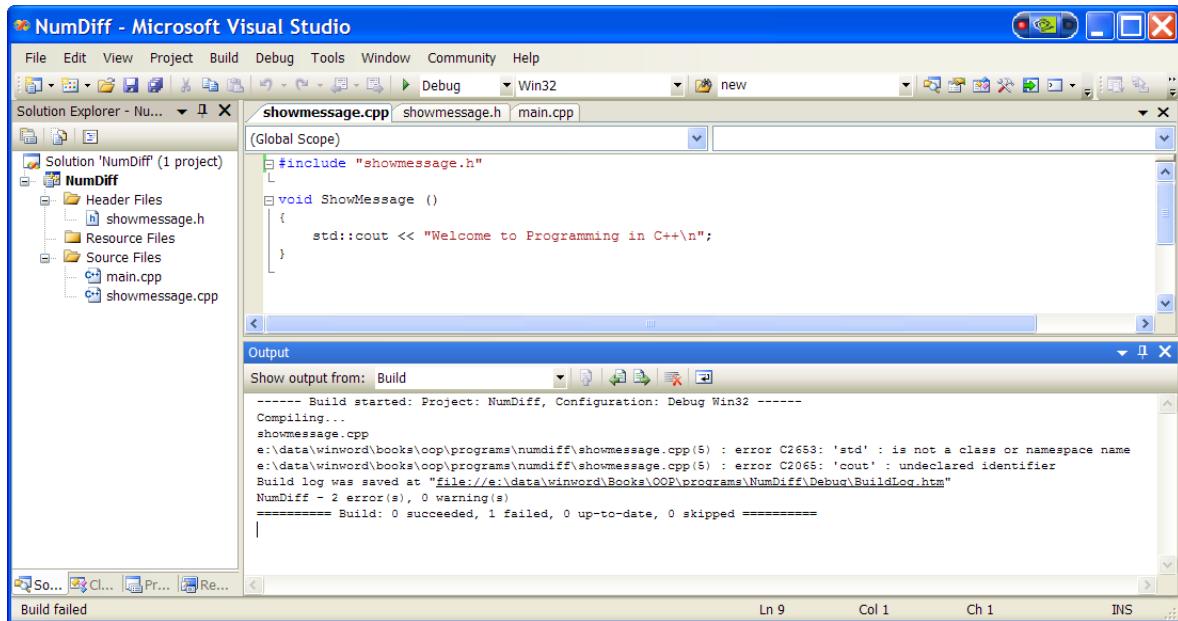


Fig. 2.1.14 Compile error messages

We will correct the error – we forgot to add the line `#include <iostream>` as the first line in `showmessage.cpp` file. Once we add the statement, save the file and build the application, everything works out as planned and we get the same output as in Fig. 2.1.10.

2.2 Building a Single Document Interface (SDI) Application

Step 1: Select an MFC Project

Launch Visual Studio. Click File, New and then Project. Make sure that Visual C++, MFC and MFC Application are selected as shown in Fig. 2.2.1.

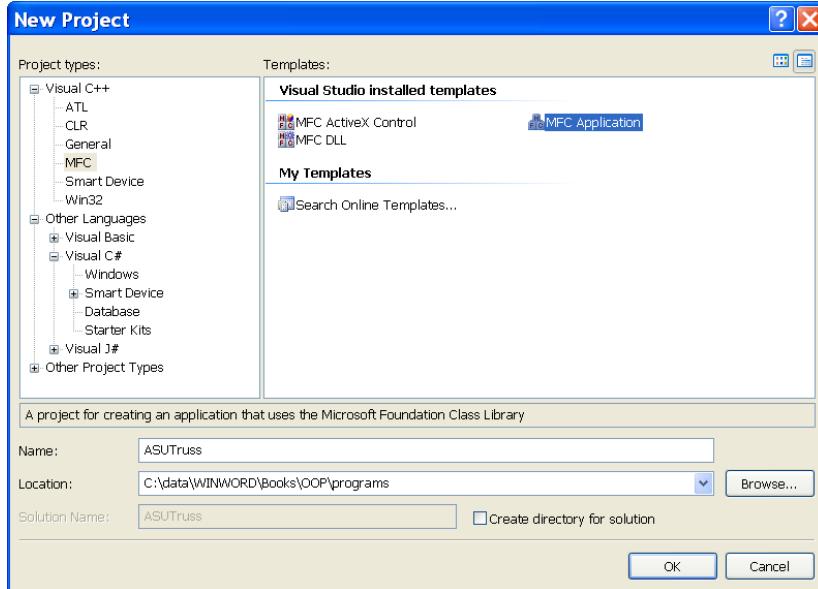


Fig. 2.2.1 Selecting a MFC Project

Type in the **Name** of the project and make sure the location points to the correct directory. Click the **OK** button. This launches the MFC Application Wizard (see Fig. 2.2.2).

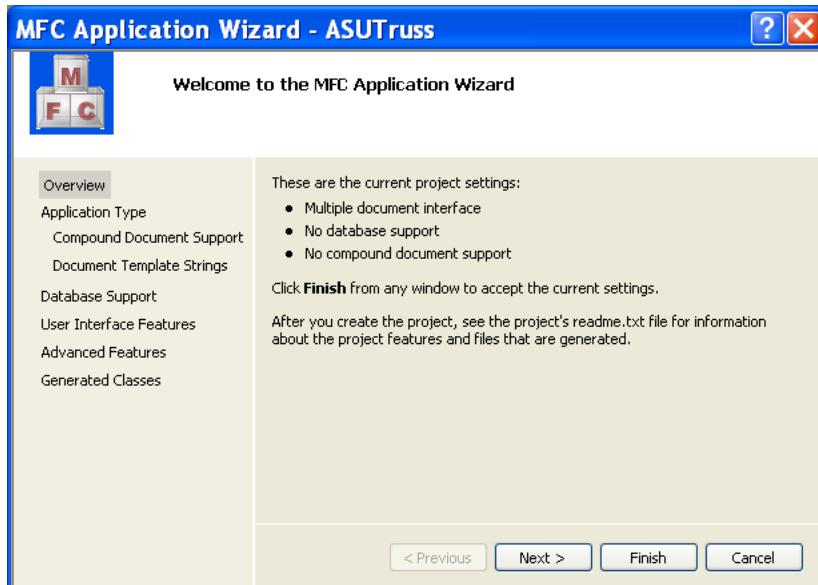


Fig. 2.2.2 MFC Application Wizard

Step 2: Specify the type of MFC Project

Follow these steps.

- (a) Click Application Type and make the selections as shown below (Fig. 2.2.3).

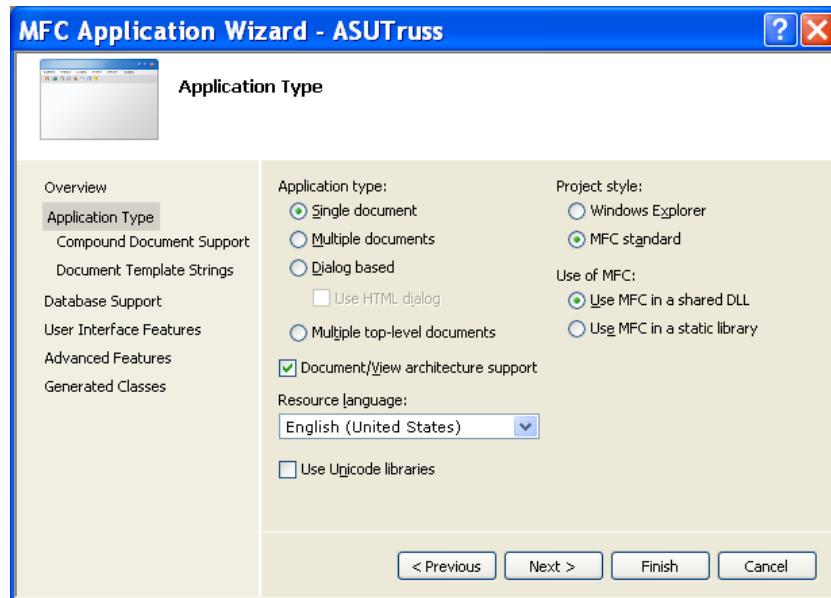


Fig. 2.2.3 Application Type selections

Note that Use Unicode libraries is not checked.

- (b) Click Document Template Strings and make the selections shown below (Fig. 2.2.4).

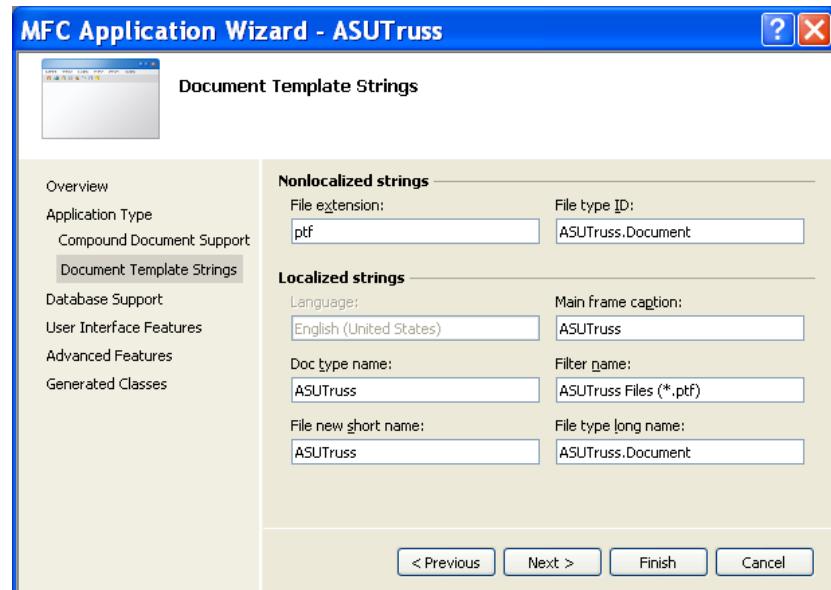


Fig. 2.2.4 Document Template Strings

(c) Click Generated Classes.

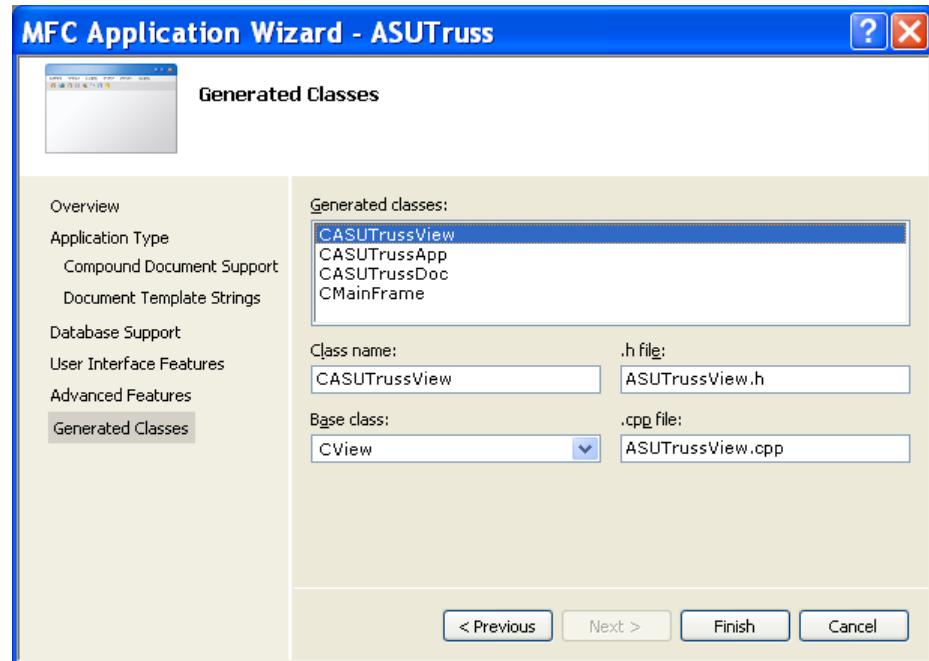


Fig. 2.2.5 List of generated classes

Click the **Finish** button.

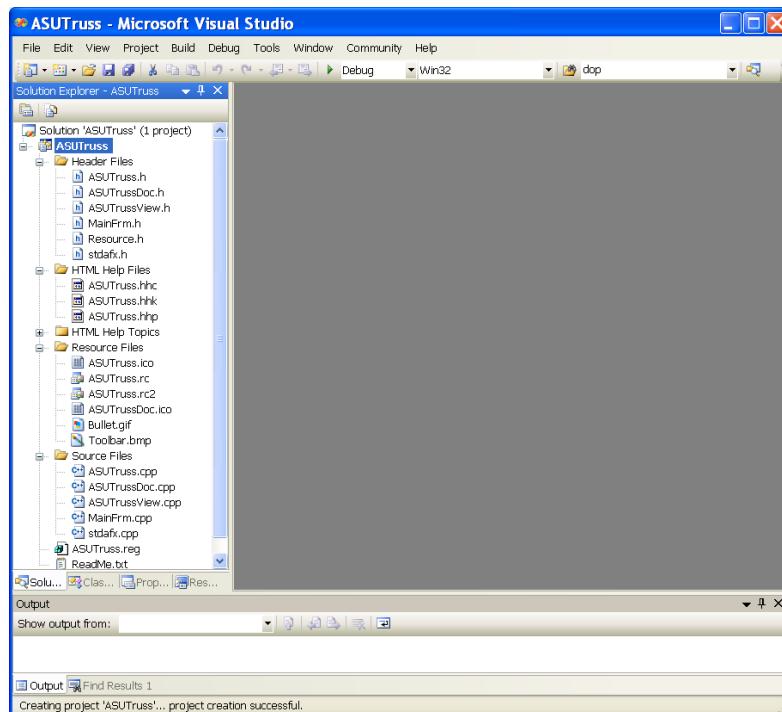


Fig. 2.2.6 MFC project ready for user customization

Step 3: Add support for File Open

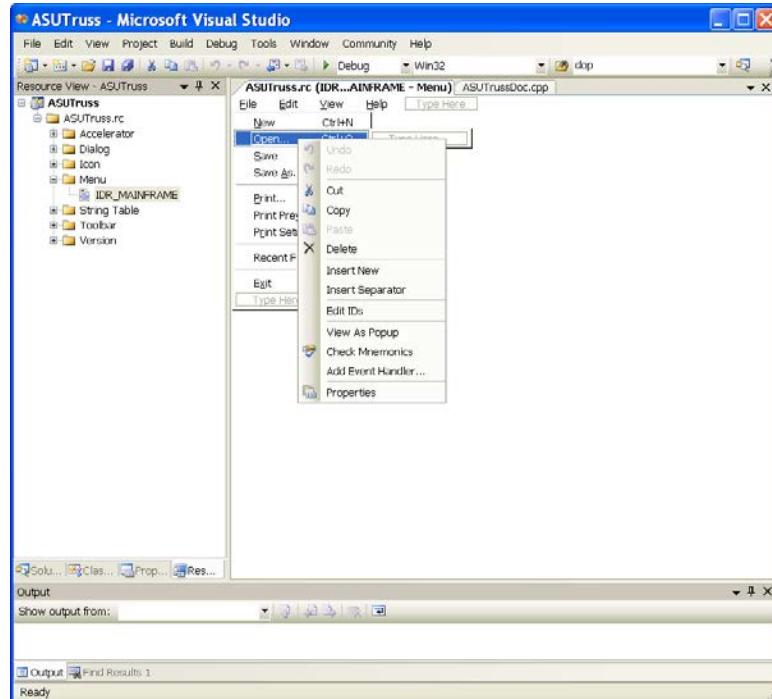


Fig. 2.2.7

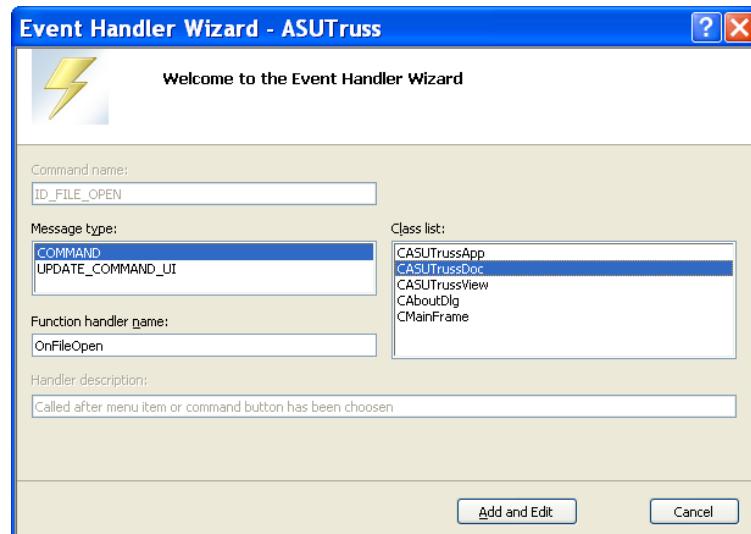


Fig. 2.2.8

3.0 Interactive Debugging

Debugging is the process of fixing errors in programs that fail to execute properly or fail to build at all. Some of these errors may be due to incorrect syntax, misspelled keywords, or incorrect type matching. These errors are called compile-time errors, and they prevent the building of programs. Once compile-time errors are fixed, the debugger can be used to locate and correct other errors in logic, sequencing, and interactions among program functions and body.

The debugging features built into the development environment enable the user to test the user's code. The user can set and manage breakpoints, and view and change variables. During the debugging process, you can execute the program command line by command line.

We will illustrate the debugging process through a modified version of the example used in Section 2.1. Before we look at the debugging process, let's look at the terminology associated with MSVS debugging.

Output Window

As we have seen before the output window normally appears at the bottom right corner. It is in this window that compile, link and possibly, error messages appear.

Breakpoints

The breakpoints set the locations where execution should be stopped to allow the user to examine the program code, variables, and to make changes, continue execution, or terminate execution. The breakpoints can be toggled on and off by the **F9** key. The debugging stage requires that at least one breakpoint is set in the code.

Action: Build the program as we discussed in Section 2.1. Double click main.cpp and bring the source code into the edit window as shown in Fig. 3.1.

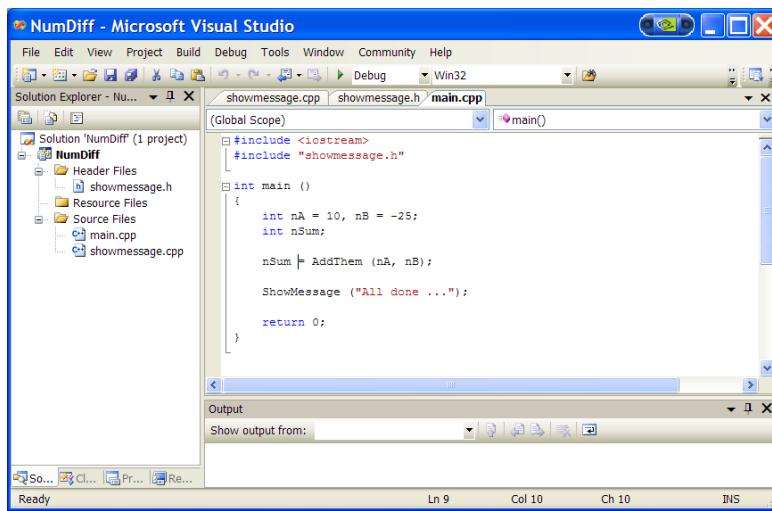


Fig. 3.1 Program ready to debug

Action: Position the cursor at line 10 and press the F9 key. A red dot appears in the grey border at the beginning of the line as shown in Fig. 3.2.

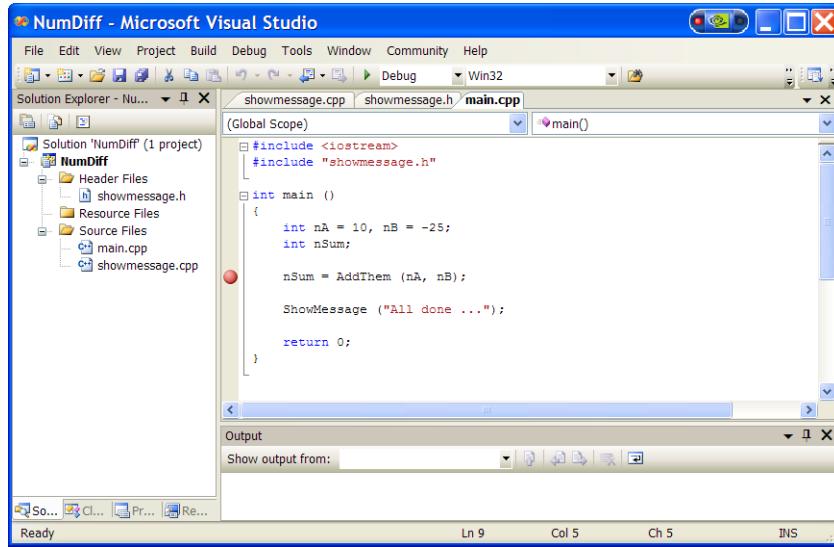


Fig. 3.2 Breakpoint set at line 10

Once the breakpoint(s) is/are set, we can start debugging by selecting one of the debug options under the *Build* menu. Fig. 3.3 shows the *Debug* menu and the shortcut keys.

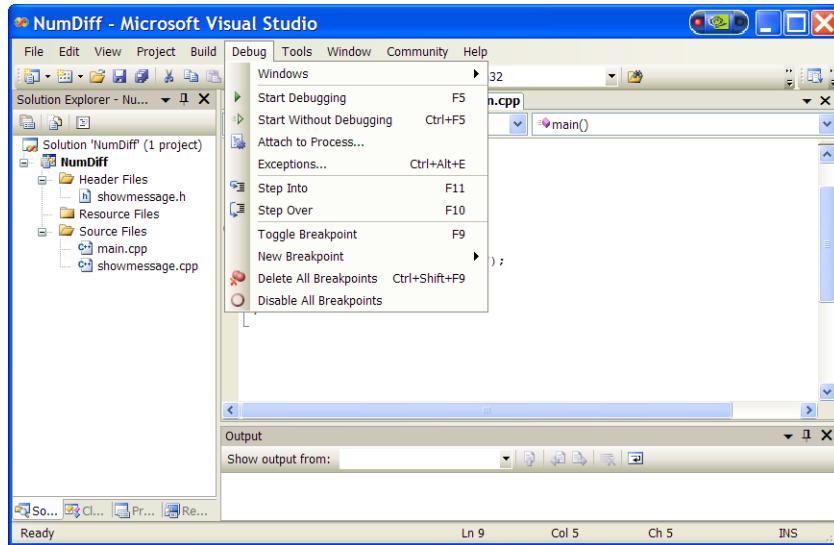


Fig. 3.3 The **Debug** menu options

The easiest way to initiate the debugging process is to press the F5 key that launches the interactive debugger.

Action: Press the F5 key. The debugger launches our program and the execution is suspended once execution reaches line 10. The resulting screen is shown in Fig. 3.4. Note that at any stage pressing the F5 key continues the debugging process until the next breakpoint is reached, or an error is encountered or the program terminates.

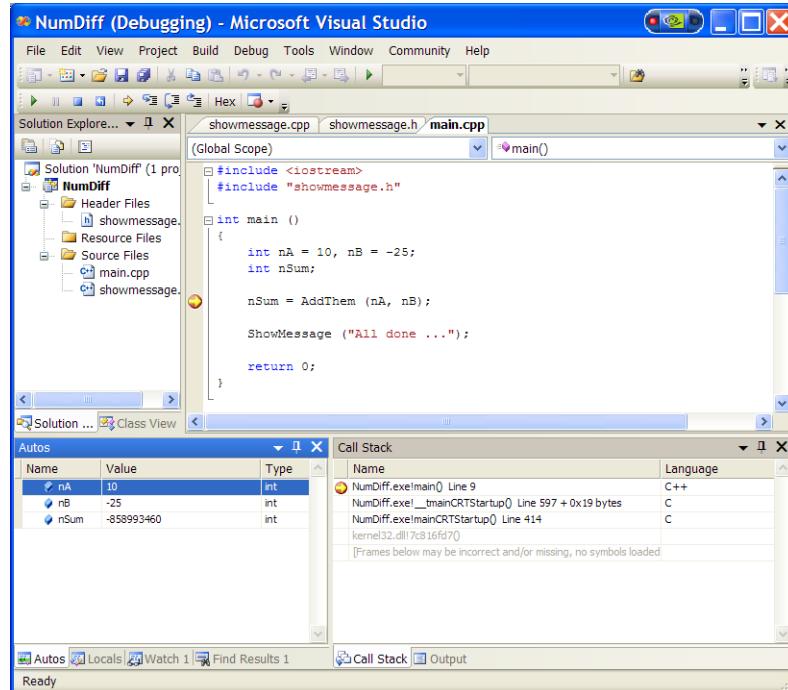


Fig. 3.4 Execution is suspended at line 10

The yellow arrow indicates which executable statement is about to be executed. Note the new windows that appear at the bottom – the Call Stack window on the right and several tabbed windows on the left. Individual tabbed windows can be activated by clicking on the tab.

Call Stack

This window shows all the functions in the stack starting with the function where execution is currently suspended.

Autos Window

The Autos window (Debug -> Windows -> Autos) will display variables and expressions from the current line of code, and the preceding line of code. Note the values of variables nA, nB and nSum (the variable is uninitialized!).

Locals Window

You can open the locals window from the Debug menu (Windows -> Locals). The locals window will automatically display all local variables in the current block of code. If you are inside of a method, the locals window will display the method parameters and locally defined variables.

Watch Window

The Watch window is used to examine user specify variables and expressions while debugging your program. You can also modify the value of a variable using this window. Just double-click on the value and type in another value.

Action: We will step into the **AddThem** function. “Step into” means that the debugging execution will step into the function and pause before the first statement in that function is executed. Press the F11 key.

The resulting screen is shown in Fig. 3.5.

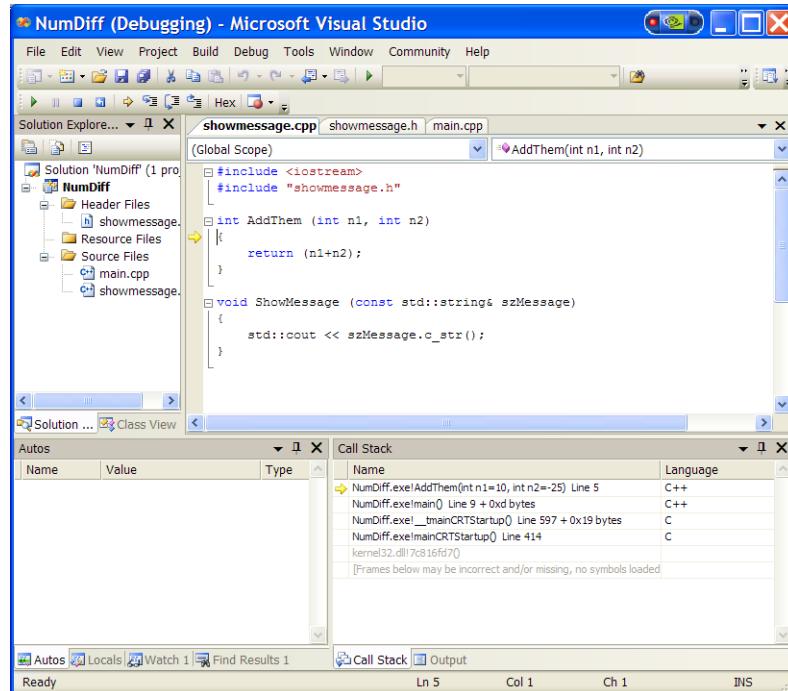


Fig. 3.5 Execution suspended at the beginning of function AddThem

Action: Press the F10 key. This key is used to step to the next statement. Note the difference between the F10 key (steps to the next statement in the same function) and the F11 key (steps into the beginning of the function). Highlight the expression $n1+n2$ and note that the debugger evaluates the expression and shows the value. Keep pressing the F10 key and trace the execution of the program till the last statement in the main program after which you should press F5 to terminate execution of the program.

4.0 Miscellaneous Topics

4.1 Specifying Command Line Arguments

Step 1: Press Alt and F7. This launches the Property Pages dialog box.

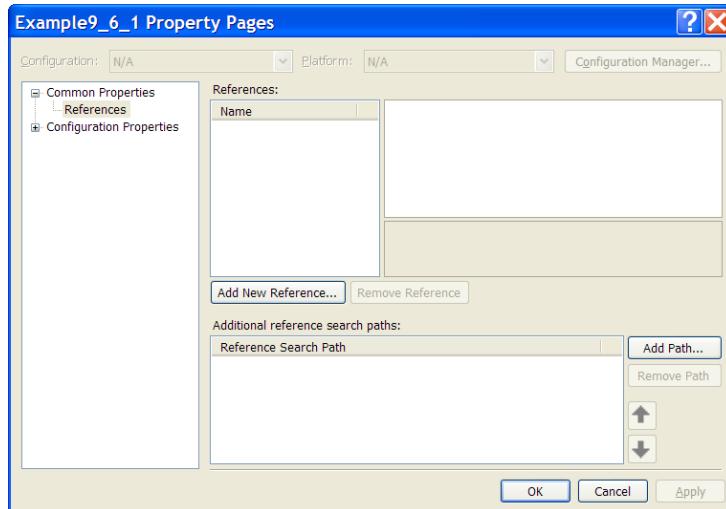


Fig. 4.1.1 Property Pages dialog box

Step 2: Expand Configuration Properties item. Click Debugging item. Now enter the command line arguments in the Command Arguments edit box as shown in Fig. 4.1.2.

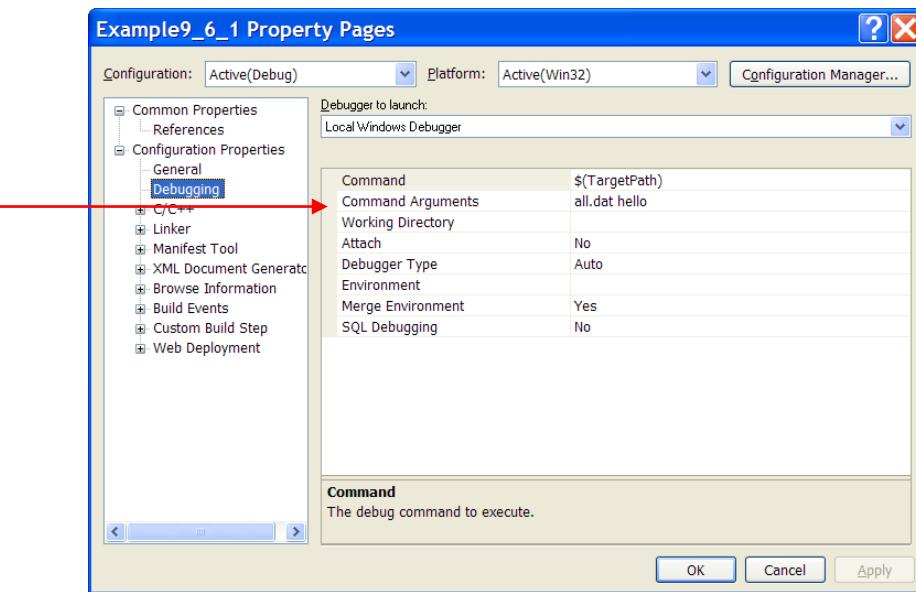


Fig. 4.1.2 **all.dat hello** as Command Arguments

4.2 Creating a Release Version

Step 1: Change the **Solution** combo-box entry to **Release**.

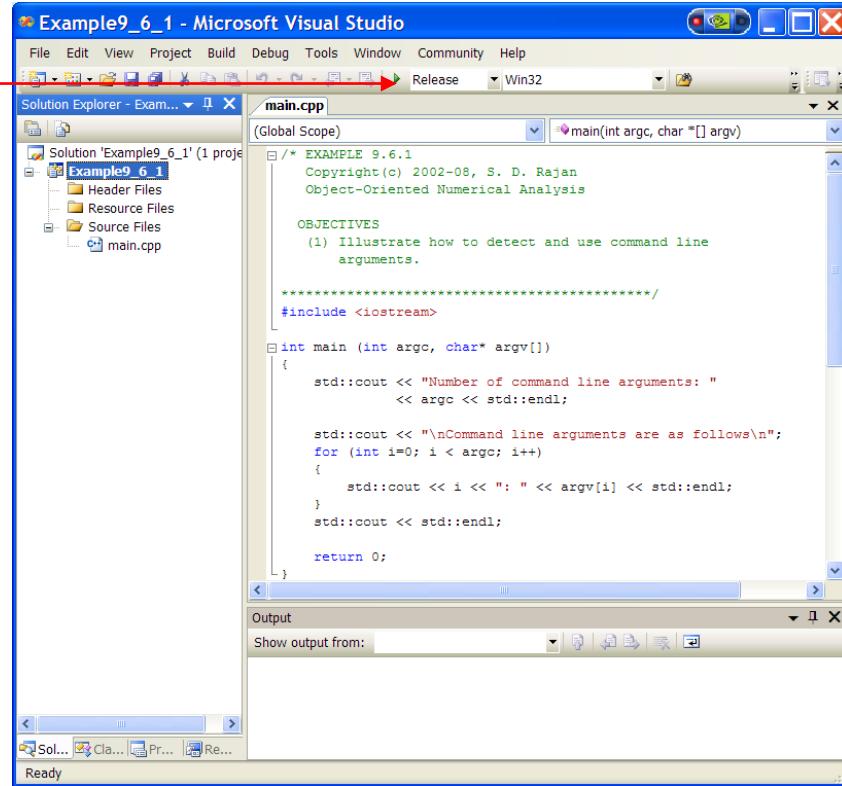


Fig. 4.2.1

Step 2: Click **Build** and select **Rebuild Solution**.

4.3 Creating a Static Library

Step 1: Click **File**, **New** and then **Project**. Make sure that Visual C++, Win32 and Win32 Project are selected. Specify the **Name** of the project, e.g. MyLib.

Step 2: In the **Application Type** select **Static library**. Uncheck **Precompiled Header**.

Step 3: Add files (either new or existing) to the project as you would with any other project.

Step 4: Click **Rebuild Solution** or **Build Solution** to create the library. Note that the default library is of the Debug type. Change the project to the Release version (as shown in Section 4.2) if you wish to build a release version of the library. A sample project is shown in Fig. 4.3.1.

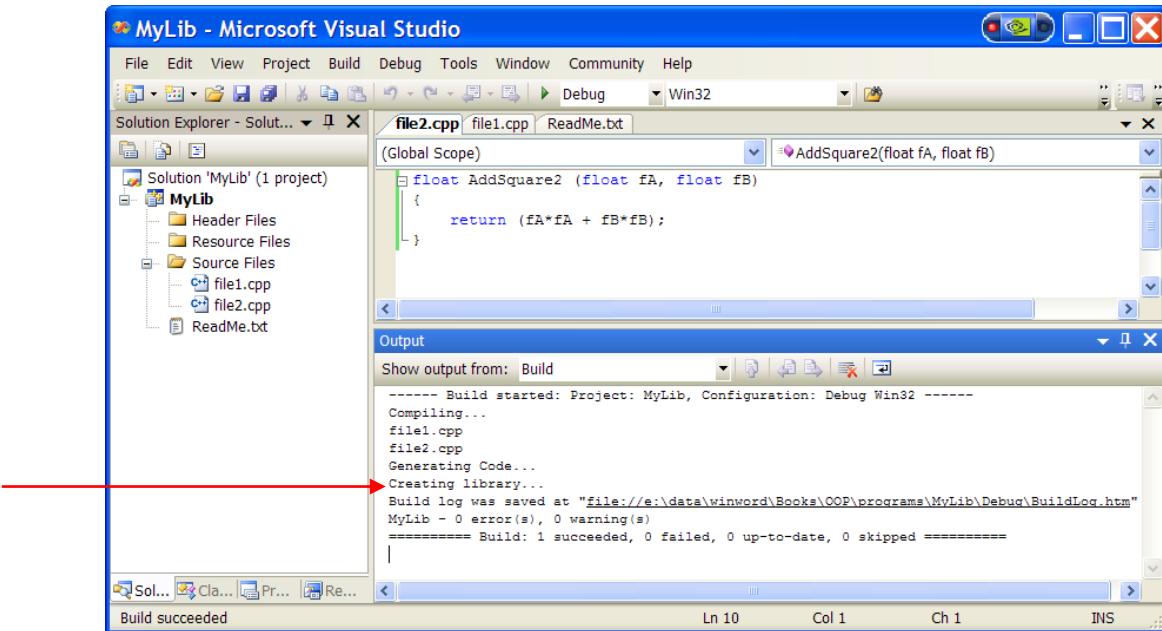


Fig. 4.3.1 Screen output showing creation of a static library

4.4 Linking with a Static Library

Step 1: Follow the Steps 1, 2 and 3 as explained in Section 2.1. A sample project is shown in Fig. 4.4.1 where the project name is UsingMyLib.

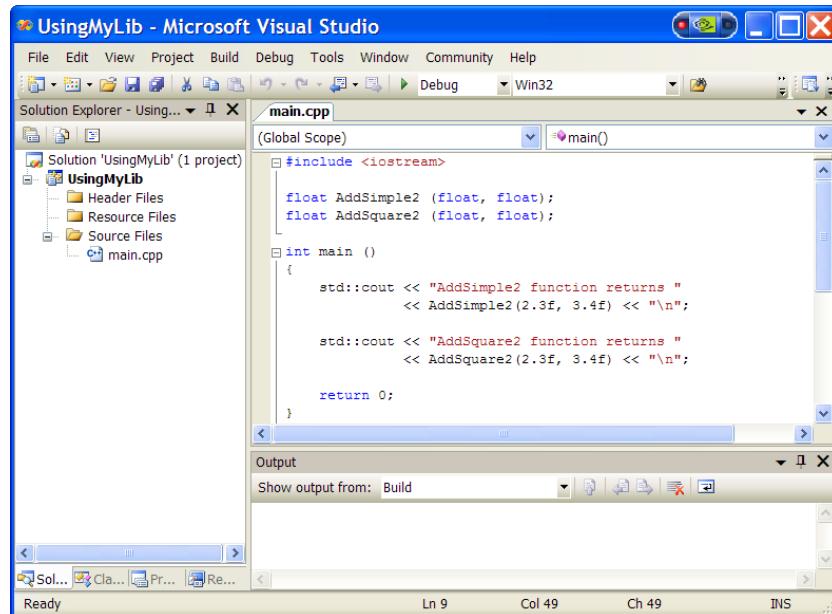


Fig. 4.4.1

Step 2: Now specify the library to link against. Click **Project**. Then select the last menu item (... Properties) where ... is the name of the project. This launches the “... Property Pages” dialog box as shown in Fig. 4.4.2. Expand **Linker**, select **Input** and type in the name of the library file as shown in the **Additional Dependencies** field.

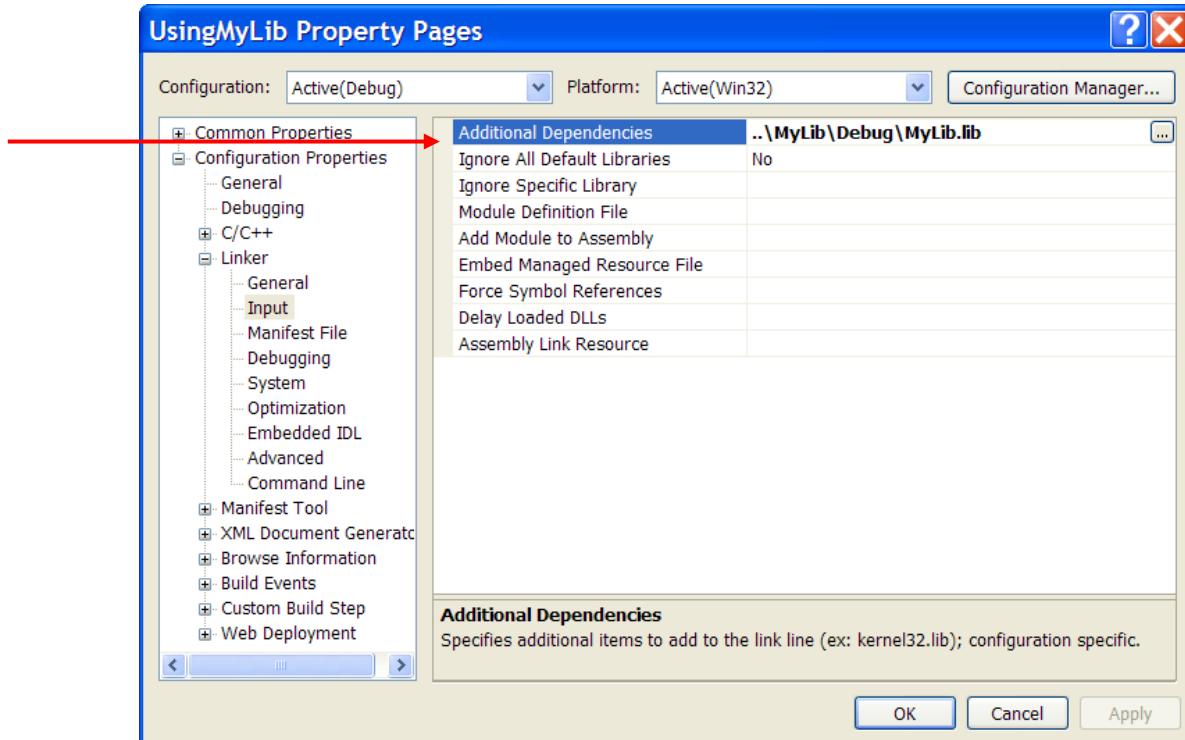


Fig. 4.4.2

Step 3: Click **Build** and then **Build Solution**. This compiles and links the program.