

# C++ and Object Oriented Numeric Computing for Scientists and Engineers

Daoqi Yang



*To Xiujuan, Joy, and Forrest*



# Preface

This book is intended to be an easy, concise, but rather complete, introduction to the ISO/ANSI C++ programming language with special emphasis on object-oriented numeric computation for students and professionals in science and engineering. The description of the language is platform-independent. Thus it applies to different operating systems such as UNIX, Linux, MacOS, Windows, and DOS, as long as a standard C++ compiler is equipped. The prerequisite of this book is elementary knowledge of calculus and linear algebra. However, this prerequisite is hardly necessary if this book is going to be used as a textbook for teaching C++ and all the sections on numeric methods are skipped. Programming experience in another language such as FORTRAN, C, Ada, Pascal, Maple, or Matlab will certainly help, but is not presumed.

All C++ features are introduced in an easy way through concepts such as functions, complex numbers, vectors, matrices, and integrals, which are familiar to every student and professional in science and engineering. In the final chapter, advanced features that are not found in FORTRAN, C, Ada, or Matlab, are illustrated in the context of iterative algorithms for linear systems such as the preconditioned conjugate gradient (CG) method and generalized minimum residual (GMRES) method. Knowledge of CG, GMRES, and preconditioning techniques is not presumed and they are explained in detail at the algorithmic level. Matrices can be stored in full (all entries are stored), band (entries outside a band along the main diagonal are zero and not stored to save memory), and sparse (only nonzero entries are stored to save memory) formats and exactly one definition for CG or GMRES is needed that is good for all three matrix storage formats. This is

in contrast to a procedural language such as FORTRAN, C, Ada, or Matlab, where three definitions of CG and GMRES may have to be given for the three matrix formats. This is one of the salient features of object-oriented programming called *inheritance*. The CG and GMRES methods are defined for a base *class* and can be inherited by classes for full, band, and sparse matrices. If one later decides to add, for example, a symmetric (full, band, or sparse) storage format (only half of the entries need be stored for a symmetric matrix to save memory), the code of CG and GMRES methods can be reused for it without any change or recompilation.

Another notable feature is generic programming through *templates*, which enables one to define a function that may take arguments of different data types at different invocations. For example, the same definition of CG and GMRES can deal with matrices with entries in single, double, and extended double (*long double*) precisions and complex numbers with different types for real and imaginary parts. Again, using C or FORTRAN would require different versions of the same function to handle different data types; when the implementation of the function is going to be changed later for robustness or efficiency, then every version has to be changed, which is tedious and error-prone. The *operator overloading* feature of C++ enables one to add and multiply matrices and vectors using + and \* in the same way as adding and multiplying numbers. The meaning of the operators is user-defined and thus it provides more flexibility than Matlab. For example, the operators + and \* can be used to add and multiply full, band, and sparse matrices. These and other features of C++ such as information hiding, encapsulation, polymorphism, error handling, and standard libraries are explained in detail in the text. With increasingly complicated numeric algorithms, many sophisticated data structures can be relatively easily implemented in C++, rather than in FORTRAN or C; this is a very important feature of C++. The C++ compiler also checks for more type errors than FORTRAN, C, Ada, Pascal, and many other languages do, which makes C++ safer to use.

However, there are trade-offs. For example, *templates* impose compile-time overhead and *inheritance* (with dynamic type binding) may impose run-time overhead. These features could slow down a program at compile- and run-times. A good C++ compiler can minimize these overheads and a programmer can still enjoy programming in C++ without suffering noticeable compile- and run-time slowdowns but with spending possibly much less time in code development. This is evidenced by the popularity of C++ in industry for database, graphics, and telecommunication applications, where people also care dearly about the speed of a language. The fact is that C++ is getting faster as people are spending more time optimizing the compiler. Some performance comparisons on finite element analysis have shown that C++ is comparable to FORTRAN 77 and C in terms of speed. On the other hand, C++ has the potential of outperforming FORTRAN 77 and C for CPU-intensive numeric computations, due to its built-in arithmetic compound operators, template mechanisms that can, for example,

avoid the overhead of passing functions as arguments to other functions, and high performance libraries (like *valarray*), which can be optimized on different computer architectures. This potential has become a reality in many test examples.

This book consists of three parts. The first part (Chapters 1 to 4) is an introduction to basic features of C++, which have equivalents in FORTRAN 90 or C. When a person knows only this part, he can do whatever a FORTRAN 90 or C programmer can do. A brief introduction is also made to *Makefile*, debuggers, making a library, and how to time and profile a program. The second part (Chapters 5 to 9) is on object-oriented and generic programming features of C++, which can not be found in FORTRAN 90 or C. Many techniques for writing efficient C++ programs are also included; see §6.5, §7.2, §7.6, §7.2.4, §7.7, and §8.6. It is this part that causes many people to think that C++ is “too complicated.” To make it easy to understand for a typical science and engineering student or professional, I make use of basic concepts and operations for complex numbers, vectors, matrices, and integrals. This should be readily acceptable by a person with elementary knowledge of calculus and matrix algebra. The third part (Chapters 10 and 11) provides a description of C++ standard libraries on containers (such as linked list, set, vector, map, stack, and queue) and algorithms (such as sorting a sequence of elements and searching an element from a sequence according to different comparison criteria), and an introduction to a user-defined numeric linear algebra library (downloadable from my Web page) that contains functions for preconditioned conjugate gradient and generalized minimum residual methods for real and complex matrices stored in full, band, and sparse formats. Gauss elimination with and without pivoting is also included for full and band matrices. This enhances the reader’s understanding of Parts 1 and 2. Furthermore, it can save the reader a great deal of time if she has to write her own basic numeric linear algebra library, which is used on a daily basis by many scientists and engineers. Great care has been taken so that features that are more easily understood and that are more often used in numeric computing are presented first. Some examples in the book are taken from or can be used in numeric computing libraries, while others are made up only to easily illustrate features of the C++ language. A Web page (<http://www.math.wayne.edu/~yang/book.htm>) is devoted to the book on information such as errata and downloading programs in the book.

On the numeric side, discussions and programs are included for the following numeric methods in various chapters: polynomial evaluation (§3.12), numeric integration techniques (§3.13, §5.1, and §7.7), vector and matrix arithmetic (§6.3 and §7.1), iterative algorithms for solving nonlinear equations (§4.7), polynomial interpolation (§7.8), iterative and direct algorithms for solving systems of linear equations in real and complex domains (§6.6, §11.3, and §11.4), Euler and Runge-Kutta methods for solving ordinary differential equations (§5.9), and a finite difference method for solving par-

tial differential equations (§11.5) with real and complex coefficients. The coverage of numeric algorithms is far from being complete. It is intended to provide a basic understanding of numeric methods and to see how C++ can be applied to program these methods efficiently and elegantly. Most of them are covered at the end of a chapter. People with an interest in learning how to program numeric methods may read them carefully, while readers who just want to learn C++ may wish to skip them or read them briefly.

C++ is not a perfect language, but it seems to have all the features and standard libraries of which a programmer can dream. My experience is that it is much easier to program than FORTRAN and C, because FORTRAN and C give too many run-time errors and have too few features and standard libraries. Many such run-time errors are hard to debug but can easily be caught by a C++ compiler.

After all, C++ is just a set of notation to most people and a person does not have to know all the features of C++ in order to write good and useful programs. Enjoy!

### **How to Use This Book:**

This book can be used as a textbook or for self-study in a variety of ways.

1. The primary intent of this book is to teach C++ and numeric computing at the same time, for students and professionals in science and engineering. C++ is first introduced and then applied to code numeric algorithms.
2. It can also be used for people who just want to learn basic and advanced features of C++. In this case, sections on numeric computing can be omitted, and knowledge of calculus and linear algebra is not quite necessary. However, in some sections, the reader should know what a vector and a matrix are, and basic operations on vectors and matrices, such as vector addition, scalar-vector multiplication, and matrix-vector multiplication.
3. This book can be used as a reference for people who are learning numeric methods, since C++ programs of many numeric methods are included. Students who learn numeric methods for the first time often have difficulty programming the methods. The C++ code in the book should help them get started and have a deeper understanding of the numeric methods.
4. For experienced C++ programmers, this book may be used as a reference. It covers essentially all features of the ISO/ANSI C++ programming language and libraries. It also contains techniques for writing efficient C++ programs. For example, standard operator overloading for vector operations, as introduced in most C++ books, may be



a few times slower than using C or FORTRAN style programming. Techniques are given in the book so that the C++ code is no slower than its corresponding C or FORTRAN style code. Examples are also given to replace certain virtual functions by static polymorphism to improve run-time efficiency. Other advanced techniques include expression templates and template metaprograms.

**Acknowledgments:**

This book has been used as the textbook for a one-semester undergraduate course on C++ and numeric computing at Wayne State University. The author would like to thank his students for valuable suggestions and discussions. Thanks also go to the anonymous referees and editors whose careful review and suggestions have improved the quality of the book. Readers' comments and suggestions are welcome and can be sent to the author via email (dyang@na-net.ornl.gov).

Daoqi Yang  
Wayne State University  
Detroit, Michigan

June 28, 2000



# Contents

<b>Preface</b>	<b>vii</b>
<b>1 Basic Types</b>	<b>1</b>
1.1 A Sample Program . . . . .	1
1.2 Types and Declarations . . . . .	6
1.3 Basic Types . . . . .	8
1.3.1 Integer Types . . . . .	8
1.3.2 Character Types . . . . .	10
1.3.3 Floating Point Types . . . . .	11
1.3.4 The Boolean Type . . . . .	14
1.3.5 The Void Type . . . . .	14
1.4 Numeric Limits . . . . .	15
1.5 Identifiers and Keywords . . . . .	19
1.5.1 Identifiers . . . . .	19
1.5.2 Keywords . . . . .	20
1.6 Exercises . . . . .	21
<b>2 Expressions and Statements</b>	<b>25</b>
2.1 Scopes and Storage Classifications . . . . .	25
2.1.1 Local and Global Variables . . . . .	26
2.1.2 External and Register Variables . . . . .	27
2.2 Expressions . . . . .	28
2.2.1 Arithmetic Expressions . . . . .	28
2.2.2 Relational Expressions . . . . .	29

2.2.3	Logical Expressions . . . . .	30
2.2.4	Bitwise Expressions . . . . .	31
2.2.5	Comma Expressions . . . . .	33
2.3	Statements . . . . .	33
2.3.1	Declarations and Initializations . . . . .	34
2.3.2	Assignments . . . . .	35
2.3.3	Compound Assignments . . . . .	35
2.3.4	Increments and Decrements . . . . .	36
2.3.5	Compound Statements . . . . .	36
2.3.6	Conditional Statements . . . . .	37
2.3.7	Iteration Statements . . . . .	41
2.4	Fibonacci Numbers . . . . .	44
2.5	Exercises . . . . .	47
<b>3</b>	<b>Derived Types</b>	<b>51</b>
3.1	Constants and Macros . . . . .	51
3.2	Enumerations . . . . .	52
3.3	Arrays . . . . .	54
3.4	Structures . . . . .	57
3.5	Unions and Bit Fields . . . . .	57
3.6	Pointers . . . . .	60
3.6.1	Pointer Arithmetic . . . . .	64
3.6.2	Multiple Pointers . . . . .	64
3.6.3	Offsetting Pointers . . . . .	67
3.6.4	Constant Pointers . . . . .	68
3.6.5	Void and Null Pointers . . . . .	70
3.6.6	Pointers to Structures . . . . .	70
3.6.7	Pointers to Char . . . . .	72
3.6.8	Pointers and Arrays . . . . .	72
3.7	References . . . . .	72
3.8	Functions . . . . .	73
3.8.1	Function Declarations and Definitions . . . . .	73
3.8.2	Function Overloading . . . . .	74
3.8.3	Argument Passing . . . . .	75
3.8.4	Return Values . . . . .	79
3.8.5	Recursive Functions . . . . .	81
3.8.6	Inline Functions . . . . .	81
3.8.7	Default Arguments . . . . .	82
3.8.8	Function Types . . . . .	83
3.8.9	Static Local Variables . . . . .	84
3.8.10	The Function <i>main</i> . . . . .	85
3.9	Program Execution . . . . .	86
3.10	Operator Summary and Precedence . . . . .	88
3.11	Standard Library on Mathematical Functions . . . . .	92
3.12	Polynomial Evaluation . . . . .	94

3.13	Trapezoidal and Simpson's Rules . . . . .	98
3.14	Exercises . . . . .	101
<b>4</b>	<b>Namespaces and Files</b>	<b>113</b>
4.1	Namespaces . . . . .	113
4.1.1	Using Declarations and Directives . . . . .	117
4.1.2	Multiple Interfaces . . . . .	119
4.1.3	Namespace Alias . . . . .	120
4.1.4	Unnamed Namespaces . . . . .	121
4.1.5	Name Lookup . . . . .	121
4.2	Include Files . . . . .	122
4.2.1	Include Files for Standard Libraries . . . . .	122
4.2.2	User's Own Include Files . . . . .	124
4.2.3	Conditional Include Directives . . . . .	126
4.2.4	File Inclusion . . . . .	128
4.3	Source Files and Linkages . . . . .	129
4.3.1	Separate Compilation . . . . .	129
4.3.2	External and Internal Linkages . . . . .	129
4.3.3	Linkage to Other Languages . . . . .	133
4.4	Some Useful Tools . . . . .	134
4.4.1	How to Time a Program . . . . .	134
4.4.2	Compilation Options and Debuggers . . . . .	136
4.4.3	Creating a Library . . . . .	138
4.4.4	Makefile . . . . .	139
4.5	Standard Library on Strings . . . . .	142
4.5.1	Declarations and Initializations . . . . .	142
4.5.2	Operations . . . . .	143
4.5.3	C-Style Strings . . . . .	144
4.5.4	Input and Output . . . . .	144
4.5.5	C Library on Strings . . . . .	145
4.6	Standard Library on Streams . . . . .	146
4.6.1	Formatted Integer Output . . . . .	146
4.6.2	Formatted Floating Point Output . . . . .	148
4.6.3	Output Width . . . . .	149
4.6.4	Input and Output Files . . . . .	150
4.6.5	Input and Output of Characters . . . . .	153
4.6.6	String Streams . . . . .	157
4.7	Iterative Methods for Nonlinear Equations . . . . .	158
4.7.1	Bisection Method . . . . .	159
4.7.2	Newton's Method . . . . .	165
4.8	Exercises . . . . .	167
<b>5</b>	<b>Classes</b>	<b>173</b>
5.1	Class Declarations and Definitions . . . . .	173
5.2	Copy Constructors and Copy Assignments . . . . .	180

5.3	Friends . . . . .	184
5.4	Static Members . . . . .	185
5.5	Constant and Mutable Members . . . . .	187
5.6	Class Objects as Members . . . . .	189
5.7	Array of Classes . . . . .	191
5.8	Pointers to Members . . . . .	193
5.9	Numeric Methods for Ordinary Differential Equations . . . . .	194
5.10	Exercises . . . . .	199
<b>6</b>	<b>Operator Overloading</b>	<b>203</b>
6.1	Complex Numbers . . . . .	204
6.1.1	Initialization . . . . .	206
6.1.2	Default Copy Construction and Assignment . . . . .	206
6.1.3	Conversions and Mixed-Mode Operations . . . . .	207
6.2	Operator Functions . . . . .	207
6.3	Vectors and Matrices . . . . .	211
6.4	Explicit and Implicit Conversions . . . . .	220
6.5	Efficiency and Operator Overloading . . . . .	223
6.6	Conjugate Gradient Algorithm . . . . .	226
6.7	Exercises . . . . .	229
<b>7</b>	<b>Templates</b>	<b>231</b>
7.1	Class Templates . . . . .	231
7.1.1	Member and Friend Definitions . . . . .	233
7.1.2	Template Instantiation . . . . .	234
7.1.3	Template Parameters . . . . .	235
7.1.4	Type Equivalence . . . . .	235
7.1.5	User-Defined Specializations . . . . .	235
7.1.6	Order of Specializations . . . . .	239
7.2	Function Templates . . . . .	240
7.2.1	Function Template Parameters . . . . .	241
7.2.2	Function Template Overloading . . . . .	242
7.2.3	Specializations . . . . .	243
7.2.4	Class Templates as Function Template Parameters . . . . .	245
7.2.5	Member Function Templates . . . . .	246
7.2.6	Friend Function Templates . . . . .	247
7.3	Template Source Code Organization . . . . .	248
7.4	Standard Library on Complex Numbers . . . . .	250
7.5	Standard Library on <i>valarrays</i> . . . . .	251
7.5.1	The Type <i>valarray</i> . . . . .	252
7.5.2	Slice Arrays . . . . .	253
7.5.3	Generalized Slice Arrays . . . . .	256
7.5.4	Mask Arrays and Indirect Arrays . . . . .	257
7.6	Standard Library on Numeric Algorithms . . . . .	258
7.6.1	Accumulate . . . . .	259

7.6.2	Inner Products . . . . .	262
7.6.3	Partial Sums . . . . .	263
7.6.4	Adjacent Differences . . . . .	263
7.7	Efficient Techniques for Numeric Integration . . . . .	264
7.7.1	Function Object Approach . . . . .	264
7.7.2	Function Pointer as Template Parameter . . . . .	265
7.7.3	Using Dot Products and Expression Templates . . . . .	266
7.7.4	Using Dot Products and Template Metaprograms . . . . .	270
7.8	Polynomial Interpolation . . . . .	273
7.8.1	Lagrange Form . . . . .	273
7.8.2	Newton Form . . . . .	275
7.9	Exercises . . . . .	279
<b>8</b>	<b>Class Inheritance</b>	<b>283</b>
8.1	Derived Classes . . . . .	283
8.1.1	Member Functions . . . . .	286
8.1.2	Constructors and Destructors . . . . .	286
8.1.3	Copying . . . . .	287
8.1.4	Class Hierarchy . . . . .	287
8.1.5	Virtual Functions . . . . .	288
8.1.6	Virtual Destructors . . . . .	289
8.2	Abstract Classes . . . . .	291
8.3	Access Control . . . . .	296
8.3.1	Access to Members . . . . .	296
8.3.2	Access to Base Classes . . . . .	297
8.4	Multiple Inheritance . . . . .	300
8.4.1	Ambiguity Resolution . . . . .	301
8.4.2	Replicated Base Classes . . . . .	302
8.4.3	Virtual Base Classes . . . . .	303
8.4.4	Access Control in Multiple Inheritance . . . . .	304
8.5	Run-Time Type Information . . . . .	305
8.5.1	The <i>dynamic_cast</i> Mechanism . . . . .	306
8.5.2	The <i>typeid</i> Mechanism . . . . .	308
8.5.3	Run-Time Overhead . . . . .	309
8.6	Replacing Virtual Functions by Static Polymorphism . . . . .	309
8.7	Exercises . . . . .	316
<b>9</b>	<b>Exception Handling</b>	<b>319</b>
9.1	Throw and Catch . . . . .	319
9.2	Deriving Exceptions . . . . .	323
9.3	Catching Exceptions . . . . .	325
9.3.1	Re-throw . . . . .	326
9.3.2	Catch All Exceptions . . . . .	326
9.3.3	Order of Handlers . . . . .	327
9.4	Specifying Exceptions in Functions . . . . .	327

9.5	Standard Exceptions . . . . .	329
9.6	Exercises . . . . .	331
<b>10</b>	<b>Standard Libraries on Containers and Algorithms</b>	<b>333</b>
10.1	Standard Containers . . . . .	333
10.1.1	Vector . . . . .	334
10.1.2	List . . . . .	341
10.1.3	Map and Set . . . . .	344
10.1.4	Stack and Queue . . . . .	347
10.2	Standard Algorithms . . . . .	348
10.2.1	Sorting, Copying, and Replacing Algorithms . . . . .	348
10.2.2	Searching and Traversing Algorithms . . . . .	355
10.2.3	Set, Permutation, and Heap Algorithms . . . . .	360
10.3	Standard Function Objects and Adaptors . . . . .	365
10.3.1	Arithmetic Function Objects . . . . .	365
10.3.2	Relational Function Objects . . . . .	366
10.3.3	Logical Function Objects . . . . .	366
10.3.4	Standard Adaptors . . . . .	367
10.4	Exercises . . . . .	368
<b>11</b>	<b>Linear System Solvers</b>	<b>371</b>
11.1	Matrix Storage Formats . . . . .	372
11.1.1	Full Matrices . . . . .	372
11.1.2	Band Matrices . . . . .	372
11.1.3	Sparse Matrices . . . . .	374
11.2	A Class Hierarchy for Matrices . . . . .	375
11.3	Iterative Algorithms . . . . .	385
11.3.1	Conjugate Gradient Method . . . . .	385
11.3.2	Generalized Minimum Residual Method . . . . .	390
11.3.3	Preconditioning Techniques . . . . .	398
11.4	Gauss Elimination . . . . .	401
11.4.1	LU Decomposition . . . . .	401
11.4.2	Gauss Elimination Without Pivoting . . . . .	406
11.4.3	Gauss Elimination with Pivoting . . . . .	408
11.5	Finite Difference Method for Partial Differential Equations	414
11.6	Exercises . . . . .	424
	<b>References</b>	<b>427</b>
	<b>Index</b>	<b>430</b>



# 1

## Basic Types

This chapter starts with a sample C++ program and then presents basic data types for integral and floating point types, and two special types called *bool* and *void*. Towards the end of the chapter, numeric limits are introduced such as the largest integer and smallest double precision number in a particular C++ implementation on a particular computer. Finally, identifiers and keywords are discussed.

### 1.1 A Sample Program

A complete program in C++ must have a function called *main*, which contains statements between braces { and }. Each statement can extend to more than one line, but must end with a semicolon. Comments must be enclosed by /\* and \*/, or preceded by // extending to the rest of the line. The first can be used for comments that stand on many lines while the second for a whole line or an end part of a line. One of them can be used to comment out the other. Comments and blank lines are ignored by the compiler and contribute only to the readability of a program.

Below is a simple program that calculates the sum of all integers between two given integers. The user is prompted to enter these two integers from the standard input stream (usually the terminal screen) represented by *cin*. The result is output to the standard output stream (usually the terminal screen) represented by *cout*. The streams *cin* and *cout* are defined in a C++ standard library called *<iostream>*. The program reads:

```

/* A sample program to illustrate some basic features of C++.
   It adds all integers between two given integers and
   outputs the sum to the screen. */

#include <iostream>      // include standard library for I/O
using namespace std;

main() {
    int n, m;           // declare n and m to be integers
    cout << "Enter two integers: \n";    // output to screen
    cin >> n >> m;      // input will be assigned to n, m

    if (n > m) {         // if n is bigger than m, swap them
        int temp = n;    // declare temp and initialize it
        n = m;           // assign value of m to n
        m = temp;        // assign value of temp to m
    }

    double sum = 0.0;    // sum has double precision

    // a loop, i changes from n to m with increment 1 each time
    for (int i = n; i <= m; i++) { // <=: less than or equal to
        sum += i;          // sum += i: sum = sum + i;
    }

    cout << "Sum of integers from " << n << " to " << m
        << " is: " << sum << '\n'; // output sum to screen
}

```

The first three lines in the program are comments enclosed by `/*` and `*/`, which are usually used for multiline comments. Other comments in the program are short ones and preceded by two slashes `//`; they can start from the beginning or middle of a line extending to the rest of it.

Input and output are not part of C++ and their declarations are provided in a header file called *iostream* that must be included at the beginning of the program using

```
#include <iostream>
```

with the sign `#` standing in the first column. The statement

```
using namespace std;
```

lets the program gain access to declarations in the *namespace std*. All standard libraries are defined in a *namespace* called *std*. The mechanism *namespace* enables one to group related declarations and definitions together and supports modular programming; see Chapter 4 for more details. Before standard C++, these two statements could be replaced by including

`<iostream.h>`. The compiler will not recognize the identifiers *cin* and *cout* if the standard library `<iostream>` is not included. Similarly, the math library `<math.h>` must be included when mathematical functions like *sin* (sine), *cos* (cosine), *atan* (arctangent), *exp* (exponential), *sqrt* (square root), and *log* (logarithm) are used. The header files for standard libraries reside somewhere in the C++ system, and a programmer normally does not have to care where. When they are included in a program, the system will find them automatically. See §4.2 for more information on how to include a file.

The symbol `<<` is called the output operator and `>>` the input operator. The statement

```
cout << "Enter two integers: \n";
```

outputs the string *"Enter two integers:"* followed by a new line to the screen, telling the user to type in, on the keyboard, two integers separated by a whitespace. Then the statement

```
cin >> n >> m;
```

causes the computer to pause while characters are entered from the keyboard and stored in memory at locations identified by *n* and *m*. It is equivalent to the following two statements.

```
cin >> n;           // first input from screen is stored in n
cin >> m;           // second input from screen is stored in m
```

Notice that a variable must be declared before it can be used. The integer variables *n* and *m* are declared by the statement:

```
int n, m;           // declare n and m to be integers
```

The character `'\n'` represents a newline. A character string like *"Sum of integers from"* must appear between double quotation marks, while a single character appear between single quotation marks such as `'A'`, `'5'` and `'\n'`. Note that `'\n'` is a single character. See §1.3.2 for more details on characters, §4.5 on strings, and §4.6 on input and output streams.

The words *int*, *double*, *if*, and *for* are reserved words in C++ and can not be used for the name of a variable. The reserved word *int* stands for integers while *double* stands for double precision floating point numbers. The variable *sum* is declared to have double precision by

```
double sum = 0.0;   // sum is initialized to 0
```

This statement not only declares *sum* to have type *double*, but also assigns it an initial value 0. A variable may not have to be initialized when it is declared.

In the *if* conditional statement

```
if (n > m) {         // if n is bigger than m, swap them
```

```

    int temp = n;    // declare temp and initialize it
    n = m;           // assign value of m to n
    m = temp;        // assign value of temp to m
}

```

the statements inside the braces will be executed if the condition  $n > m$  ( $n$  is strictly larger than  $m$ ) is true and will be skipped otherwise. This *if* statement instructs the computer to swap (interchange) the values of  $n$  and  $m$  if  $n$  is larger than  $m$ . Thus, after this *if* statement,  $n$  stores the smaller of the two input integers and  $m$  stores the larger. Notice that a temporary variable *temp* is used to swap  $n$  and  $m$ . The braces in an *if* statement can be omitted when there is only one statement inside them. For example, the following two statements are equivalent.

```

if (n > m) m = n + 100;    // one statement
if (n > m) {               // an equivalent statement
    m = n + 100;
}

```

The second statement above can also be written in a more compact way:

```

if (n > m) { m = n + 100; }

```

The *for* loop in the sample program

```

for (int i = n; i <= m; i++) {
    sum += i;
}

```

first declares variable  $i$  of type *int* and assigns the value of  $n$  to  $i$ . If the value of  $i$  is less than or equal to the value of  $m$  (i.e.,  $i \leq m$ ), the statement inside the braces  $sum += i$  will be executed; Then statement  $i++$  (equivalent to  $i = i + 1$  here) is executed and causes the value of  $i$  to be incremented by 1. The statement  $sum += i$  (equivalent to  $sum = sum + i$ , meaning  $sum$  is incremented by the value of  $i$ ) is executed until the condition  $i \leq m$  becomes false. Thus this *for* loop adds all integers from  $n$  to  $m$  and stores the result in the variable  $sum$ . The braces in a *for* loop can also be omitted if there is only one statement inside them. For example, the *for* loop above can also be written in a more compact way:

```

for (int i = n; i <= m; i++) sum += i;

```

Except for possible efficiency differences, this *for* loop can be equivalently written as

```

for (int i = n; i <= m; i = i + 1) sum = sum + i;

```

The compound operators in  $i++$  and  $sum += i$  can be more easily optimized by a compiler and more efficient than  $i = i + 1$  and  $sum = sum + i$ . For example, an intermediate value of  $i + 1$  is usually obtained and

stored and then assigned to  $i$  in  $i = i + 1$ ; while in  $i++$  such an intermediate process could be omitted and the value of  $i$  is just incremented by 1. Notice that  $+=$  is treated as one operator and there is no whitespace between  $+$  and  $=$ . See §2.3.7 for details on the *for* statement and §2.3.3 for compound operators.

Finally the value of *sum* is output to the screen by the statement:

```
cout << "Sum of integers from " << n << " to " << m
      << " is: " << sum << '\n';    // output sum to screen
```

First the character string "*Sum of integers from* " is output, then the value of  $n$ , and then string "*to* ", value of  $m$ , string "*is:* ", value of *sum*, and finally a newline character ' $\backslash n$ '.

Suppose the above program is stored in a file called *sample.cc*. To compile it, at the UNIX or Linux command line, type

```
c++ -o add sample.cc
```

Then the machine-executable object code is written in a file called *add*. If you just type

```
c++ sample.cc
```

the object code will be automatically written in a file called *a.out* in UNIX (or Linux) and *sample.exe* in DOS. On other operating systems, compiling or running a program may just be a matter of clicking some buttons. Now you may type the name of the object code, namely, *add* or *a.out* at the UNIX command line, and input 1000 and 1. You shall see on the screen:

```
Enter two integers:
1000 1
Sum of integers from 1 to 1000 is: 500500
```

In this run, the input number 1000 is first stored in  $n$  and 1 in  $m$ . Since  $n$  is larger than  $m$ , the *if* statement interchanges the values of  $n$  and  $m$  so that  $n = 1$  and  $m = 1000$ . The *for* loop adds all integers from 1 to 1000. Here is another run of the program:

```
Enter two integers:
1 1000
Sum of integers from 1 to 1000 is: 500500
```

In the second run, the input number 1 is stored in  $n$  and 1000 in  $m$ . Since  $n$  is smaller than  $m$ , the *if* statement is skipped.

A user may not have to input values from the terminal screen. Alternatively, this sample program can also be written as

```
#include <iostream>    // include input/output library
using namespace std;
```

```

main() {
    int n = 1;           // declare integer n with value 1
    int m = 1000;        // declare integer m with value 1000

    double sum = 0;
    for (int i = n; i <= m; i++) sum += i;
    cout << "The sum is: " << sum << '\n';
}

```

or in a more compact form:

```

#include <iostream>
using namespace std;

main() {
    double sum = 0;
    for (int i = 1; i <= 1000; i++) sum += i;
    cout << "The sum is: " << sum << '\n';
}

```

The disadvantage of the alternative forms is that, if one wishes to add integers from 2 to 5050, for example, then the program needs to be modified and recompiled.

By convention, C++ files are usually postfixed with *.cc*, *.c*, *.C*, *.cpp*, and the like, and C++ compilers are *c++*, *g++*, *gcc*, *CC*, and so on.

Notice that unlike in C or FORTRAN, declarations (like *double sum;*) may not have to appear at the beginning of a program and can be mixed with other statements. The rule of thumb is that a variable should not be declared until it is used. This should increase the readability of a program and avoid possible misuse of variables.

## 1.2 Types and Declarations

Consider the mathematical formula:

$$z = y + f(x);$$

For this to make sense in C++, the *identifiers* *x*, *f*, *y*, and *z* must be suitably declared. All identifiers must be declared before they can be used. *Declarations* introduce entities represented by identifiers and their types (e.g. *int* or *double*) into a C++ program. The types determine what kind of operations can be performed on them and how much storage they occupy in computer memory. For example, the declarations

```

int x;           // x is declared to be of integer type
float y = 3.14;  // y is a floating point number
double z;        // z is a floating point number

```

```
double f(int);           // f is a function taking an integer as
                        // its argument and returning a double
```

will make the above formula meaningful, where  $+$  is interpreted as adding two numbers,  $=$  assigns the value on its right-hand side to the variable on its left-hand side, and  $f(x)$  is interpreted as a function call returning the function value corresponding to argument  $x$ . The first declaration above introduces variable  $x$  to be of type *int*. That is,  $x$  can only store integers. The second declaration introduces variable  $y$  to be of type *float* (single precision floating point number) and assigns an initial value 3.14 to it. A simple function definition is:

```
double f(int i) {
    return (i*i + (i-1)*(i-1) + (i-2)*(i-2) - 5)/3.14;
}
```

The function  $f()$  takes integer  $i$  as input and returns a double precision number as output. It calculates the value of the mathematical expression  $(i^2 + (i-1)^2 + (i-2)^2 - 5)/3.14$  for a given integer  $i$ . In C++, the symbols  $+$ ,  $-$ ,  $*$ , and  $/$  mean addition, subtraction, multiplication, and division, respectively. All these statements can be organized into a complete C++ program:

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

double f(int i) {           // function definition
    return (i*i + (i-1)*(i-1) + (i-2)*(i-2) - 5)/3.14;
}

main() {
    int x = 4;
    float y = 3.14;
    double z = y + f(x);
    cout << "The value of z is: " << z << '\n';
}
```

Note that the definition of the function  $f()$  can not be put inside the function *main()*. See §3.8 for more details on functions.

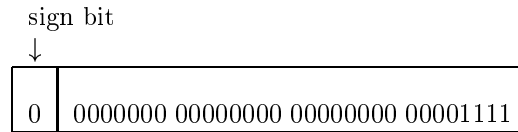
Basic types are discussed in the next section. Additional types including functions and user-defined types are discussed in subsequent chapters.

## 1.3 Basic Types

### 1.3.1 Integer Types

The integer types are *short int*, *int*, and *long int*. An *int* is of natural size for integer arithmetic on a given machine. Typically it is stored in one machine word, which is 4 bytes (32 bits) on most workstations and mainframes, and 2 bytes (16 bits) on many personal computers. A *long int* normally occupies more bytes (thus stores integers in a wider range) and *short int* fewer bytes (for integers in a smaller range).

Integers are stored in binary bit by bit in computer memory with the leading bit representing the sign of the integer: 0 for nonnegative integers and 1 for negative integers. For example, the nonnegative *int* 15 may be represented bit by bit on a machine with 4 bytes for *int* as



Note that whitespaces were inserted to easily see that it occupies 4 bytes, and the leading (leftmost) bit 0 signifies a nonnegative integer. Negative integers are normally represented in a slightly different way, called two's complement; see §2.2.4. Thus a computer with 32 bits for *int* can only store integers

$$-2^{31}, -2^{31} + 1, \dots, -2, -1, 0, 1, 2, \dots, 2^{31} - 2, 2^{31} - 1,$$

while a computer with 16 bits for *int* can only store integers

$$-2^{15}, -2^{15} + 1, \dots, -2, -1, 0, 1, 2, \dots, 2^{15} - 2, 2^{15} - 1.$$

Half of them are negative and the other half are nonnegative. Note that  $2^{31} = 2147483648$  and  $2^{15} = 32768$ . Numbers out of the given range of integers on a machine will cause *integer overflow*. When an integer overflows, the computation typically continues but produces incorrect results (see Exercises 1.6.9 and 2.5.14 for two examples). Thus a programmer should make sure integer values are within the proper range. Use *long int* if necessary or other techniques (see Exercise 3.14.21 where digits of a large integer are stored in an array of integers) for very large integers. For example, a *long int* may occupy 6 bytes and be able to store integers:

$$-2^{47}, -2^{47} + 1, \dots, -2, -1, 0, 1, 2, \dots, 2^{47} - 2, 2^{47} - 1.$$

Although the number of 8-bit bytes of storage for *short int*, *int*, and *long int* is machine-dependent, it is always given by the C++ operator *sizeof*. In general

$$\text{sizeof}(\text{short int}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long int}).$$



On one machine, `sizeof(int) = 4` and `sizeof(long int) = 6`, while on another, `sizeof(int) = sizeof(long int) = 4`. However, it is guaranteed that a *short int* has at least 16 bits and a *long int* has at least 32 bits.

A new type called *unsigned int* for nonnegative integers does not store the sign bit and thus stores larger integers than the plain *int*, which is also called *signed int*. If on a machine *signed int* holds integers from  $-32768$  to  $32767$ , then *unsigned int*, occupying the same number of bits but without storing the sign bit, will hold nonnegative integers from  $0$  to  $65535$ . An integral type (*short int*, *int*, and *long int*) can be *signed* or *unsigned*. The keyword *long* is a synonym for *long int*, *short* for *short int*, *signed* for *signed int*, and *unsigned* for *unsigned int*. An *int* is always *signed*; that is, *int* and *signed int* always mean the same thing.

To know their number of bytes on your machine, compile and run the program:

```
#include <iostream>
using namespace std;
main() {
    cout << "number of bytes in short = "
          << sizeof(short) << '\n';
    cout << "number of bytes in long int = "
          << sizeof(long) << '\n';
    cout << "number of bytes in int = "
          << sizeof(int) << '\n';
    cout << "number of bytes in unsigned int = "
          << sizeof(unsigned int) << '\n';
}
```

From now on, statements such as

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

may not be explicitly included in example programs to save space and concentrate on more important features.

Standard conversions are performed when different types appear in arithmetic operations. Truncations occur when there is not enough space for converting one type into another. For example,

```
int i = 321;           // stored in sizeof(int) bytes
short ii = 321;        // stored in sizeof(short) bytes
long iii = i;          // implicit conversion from int to long
iii = long(i);         // explicit conversion from int to long
iii = i + ii;          // implicit conversion,
                       // it is same as iii = long(i) + long(ii)

iii = 123456789;       // a big integer
```

```
ii = short(iii);    // conversion from long to short
cout << ii;         // on one machine, output of ii is -13035
```

Note that explicit type conversion requires the use of the name of the type. For example, *long(i)* converts explicitly an *int i* to *long*. Explicit type conversion is also called *cast*. Notice that when the value of long integer *iii* = 123456789 is assigned to short integer *ii*, the space occupied by *ii* (*sizeof(short)* bytes) may not be enough to hold the value of *iii*. Truncation may occur and lead to errors. For example, on one machine with *sizeof(short)* = 2, this results in *ii* = -13035. The negative sign is caused by the leading (leftmost) bit in the binary representation of *ii* (after overflow), which happens to be 1.

The suffix *U* is used to explicitly write unsigned integer constants and *L* for long integer constants. For example, 5 is a *signed int*, 5*U* is an *unsigned int* and 5*L* is a *long int*. They may occupy different numbers of bytes in memory.

```
unsigned int i1 = 5U;    // 5U means unsigned integer 5
long int i2 = 5L;       // 5L means long integer 5
int i = 5;              // 5 means signed integer 5
```

By default, a number is a decimal number (base 10). A number preceded by 0 (zero) is octal (base 8) and a number preceded by 0*x* is hexadecimal (base 16). The letters *a, b, c, d, e*, and *f*, or their upper case equivalents, represent hexadecimal numbers 10, 11, 12, 13, 14, and 15, respectively. For example, 63 is a decimal number, 077 is an octal number, and 0*x3f* is a hexadecimal number. Octal and hexadecimal numbers can be conveniently used to express bit patterns; see §6.4 for an example. See §4.6 on how to print them out.

### 1.3.2 Character Types

A *char* variable is an integral type and is of natural size to represent a character (in English and other similar languages) appearing on a computer keyboard (it occupies one byte almost universally and is assumed always to be one byte in this book). A *char* can only store an integer that fits in one byte (thus one of  $2^8 = 256$  values); it corresponds to a character in a character set, including (American) ASCII. For example, the integral value 98 corresponds to the character constant '*b*' in the ASCII character set. Character constants should appear between single quotation marks like '*A*', '*5*', '*d*', '*\n*' (newline character), '*\t*' (horizontal tab), '*\v*' (vertical tab), '*\0*' (null character), '*\\*' (backslash character), '*\"*' (double quotation character), and '*\'*' (single quotation character). Notice the special characters above that use a backslash *\* as an escape character. See Exercise 1.6.5 for a few more. For example,

```
char cc = 'A';        // assign character 'A' to cc.
```

```

// In ASCII, cc = 65.
cc = '\n';           // assign a new value to cc.
// In ASCII, cc = 10
int i = cc;           // i = 10, implicit type conversion
short ii = short(cc); // ii = 10, explicit type conversion

```

A *char*, occupying 8 bits, can range from 0 to 255 or from -128 to 127, depending on the implementation. Fortunately, C++ guarantees that a *signed char* ranges at least from -128 up to 127 and an *unsigned char* at least from 0 up to 255. The types *char*, *signed char*, and *unsigned char* are three distinct types and the use of *char* could cause portability problems due to its implementation dependency. For example,

```

char c = 255;           // c has all 8 bits 1
int i = c;              // or: int i = int(c). Now i = ?

```

What is the value of *i* now? The answer is undefined. On an SGI Challenge machine, a *char* is unsigned so that *i* = 255. On a SUN SPARC or an IBM PC, a *char* is signed so that *i* = -1. A signed integer with all bits equal to 1 (including the sign bit) represents the integer -1 in two's complement; see §2.2.4 for more details. Thus a *char* should be used primarily for characters, instead of small integers.

A *char* is output as a character rather than as a numeric value. For example, the program segment

```

char c = 'A';
int i = 'A';           // i = 65 in ASCII
cout << c << 'B' << i << "CD" << '\n' ;

```

outputs AB65CD to the screen (assuming the ASCII character set is used).

Inherited from the C programming language, a constant string always ends with a null character. For example, the string "CD" consists of three characters: C, D, and the null character (\0), and "\n" (notice the double quotation marks) is a string of two characters: the newline character (\n) and the null character (\0). In contrast, '\n' (notice the single quotation marks) is a single character. This can be checked by the *sizeof* operator:

```

int i = sizeof("CD");           // i = 3
int j = sizeof("\n");           // j = 2
int k = sizeof('\n');           // k = 1

```

### 1.3.3 Floating Point Types

The floating point types are *float*, *double*, and *long double*, which correspond to single precision, double precision, and an extended double precision, respectively. The number of 8-bit bytes of storage for each of them is given by the operator *sizeof*. In general

$$\text{sizeof}(\text{float}) \leq \text{sizeof}(\text{double}) \leq \text{sizeof}(\text{long double}).$$

On many machines, they occupy 4, 8, and 12 bytes, respectively, and a *float* stores about 6 decimal places of precision, a *double* stores about 15 decimal places, and a *long double* about 18. A *long double* with 16 bytes can have 33 decimal places of precision. The possible values of a floating point type are described by *precision* and *range*. The *precision* describes the number of significant decimal digits that a floating point number carries, while the *range* describes the largest and smallest positive floating values that can be taken by a variable of that type. For example, on machines with 4 bytes for *floats*, 8 bytes for *doubles*, and 12 bytes for *long doubles*, the range of *float* is roughly from  $10^{-38}$  to  $10^{38}$ , the range of *double* is roughly from  $10^{-308}$  to  $10^{308}$ , and the range of *long double* from  $10^{-4932}$  to  $10^{4932}$ . This means that, on such machines, a *float* number  $f$  is roughly represented in the form

$$f = \pm 0.d_1 d_2 \cdots d_6 \times 10^n,$$

where  $-38 \leq n \leq 38$  and  $d_1 \neq 0$  (here  $0.d_1 d_2 \cdots d_6$  is called the fractional part of  $f$ ); a *double* number  $d$  is roughly represented in the form

$$d = \pm 0.d_1 d_2 \cdots d_{15} \times 10^n,$$

where  $-308 \leq n \leq 308$  and  $d_1 \neq 0$ ; and a *long double* number  $g$  in the form

$$g = \pm 0.d_1 d_2 \cdots d_{18} \times 10^n,$$

where  $-4932 \leq n \leq 4932$  and  $d_1 \neq 0$ . On such machines, an *overflow* happens when a *float* number  $f = \pm q \times 10^m$  ( $0.1 \leq q < 1$ ) with  $m > 38$  and an *underflow* happens for  $m < -38$ . Similar definitions of overflow and underflow can be made for double and extended double precisions. The value of a variable is often set to zero when *underflow* occurs. The preceding discussion is true only roughly and can be made more precise in binary representation, which is normally used in the computer; see §1.4.

The numbers of bytes given as examples above are used to store floating point types. The IEEE (Institute for Electric and Electronic Engineers) standards currently require that at least 80 bits be used for doing internal computation (128 bits are used on some machines for *long double* calculations). When the results of a computation are stored or output, accuracy is often lost due to roundoff errors. For example, the statements

```
float fpi = atan(1)*4;           // include <math.h>
double dpi = atan(1)*4;
long double ldpi = atan(1)*4;
```

give different precisions of the value  $\pi$ . Note the function call *atan*(1) (for computing the arctangent of 1) gives the value  $(\pi/4)$  of the mathematical function *arctan*( $x$ ) with  $x = 1$ . Trigonometric and other functions are in the library *<math.h>*; see §3.11. To see their difference when being output

to the screen, we can use the *precision* function to specify the accuracy and use the *width* function to specify the number of digits in the output (see §4.6 for more details). For example, the statements

```
cout.precision(30);           // include <iostream>
cout.width(30);               // output occupies 30 characters
cout << fpi << '\n';
cout.width(30);
cout << dpi << '\n';
cout.width(30);
cout << ldpi << '\n';
```

output the following to the screen (on my computer).

```
3.1415927410125732421875
3.14159265358979311599796346854
3.14159265358979323851280895941
```

Compared to the exact value of  $\pi$ : 3.14159265358979323846264338..., the *float* value *fpi* has 7-digit accuracy, the *double* value *dpi* has 16-digit, while the extended double value *ldpi* has 19-digit accuracy. They are all calculated using the same formula  $\text{atan}(1) * 4$ . The internal calculation is done in the same way and accuracy is lost when the result is stored in *fpi*, *dpi*, and *ldpi*.

The IEEE standard introduces two forms of infinity (*Inf* or *Infinity* in computer output):  $+\infty$  and  $-\infty$ , for very large and small floating point numbers (when it makes sense to do so). For example,  $x/0$  and  $y+y$  give  $\infty$  for a positive floating point number  $x$  within range and the biggest floating point number  $y$ . So do  $x + \infty$ ,  $x * \infty$ , and  $\infty/x$ . Here  $\infty$  is understood to be  $+\infty$ . Similar explanations hold for  $-\infty$ . The standard also introduces the entity *NaN* (Not-a-Number) to make debugging easier. Indeterminate operations such as  $0.0/0.0$  and  $\infty - \infty$  result in *NaN*. So does  $x + NaN$ . Run the program in Exercise 1.6.12 and you will see a situation where *Infinity* and *NaN* are among the output, which can be useful debugging information.

Variables of different types can be mixed in arithmetic operations and implicit and explicit conversions can be performed. For example,

```
double d = 3.14;
int n = 2;
int m = d + n;           // m = 5, implicit type conversion
int k = int(d) + n;      // k = 5, explicit type conversion
double c = d + n;        // c = 5.14, implicit type conversion
```

Note that in the addition  $m = d + n$ , integer  $n$  is first implicitly promoted to floating point number 2.0 and the result  $d + n = 5.14$  is then implicitly truncated into integer 5 for  $m$ , causing loss of accuracy.

By default, a floating point constant is of double precision. For example, the number 0.134, 1.34E-1, 0.0134E1, or 0.0134e1 is taken to be a *double*, which occupies `sizeof(double)` bytes. Its *float* representation is 0.134F, or 0.134f, suffixed with *F* or *f*, which occupies `sizeof(float)` bytes. The number after the letter *E* or *e* means the exponent of 10. For example, 1.34e-12 means  $1.34 \times 10^{-12}$  and 1.34e12 means  $1.34 \times 10^{12}$ . This is the so-called *scientific notation* of real numbers.

The reason for providing several integer types, unsigned types, and floating point types is that there are significant differences in memory requirements, memory access times, and computation speeds for different types and that a programmer can take advantage of different hardware characteristics. The type *int* is most often used for integers and *double* for floating point numbers. Despite the fact that *float* usually requires less memory storage than *double*, arithmetics in *float* may not always be significantly or noticeably faster than in *double* for a particular C or C++ compiler, especially on modern computers; see §4.4.1 for an example and some explanations.

### 1.3.4 The Boolean Type

A Boolean, represented by the keyword *bool*, can have one of the two values: *true* and *false*, and is used to express the results of logical expressions (§2.2.3). For example,

```
bool flag = true;           // declare flag to be of bool

// ... some other code that might change the value of flag
double d = 3.14;
if (flag == false) d = 2.718;
```

The operator `==` tests for equality of two quantities. The last statement above means that, if *flag* is equal to *false*, assign 2.718 to variable *d*.

By definition, *true* has the value 1 and *false* has the value 0 when converted to an integer. Conversely, nonzero integers can be implicitly converted to *true* and 0 to *false*. A *bool* variable occupies at least as much space as a *char*. For example,

```
bool b = 7;      // bool(7) is converted to true, so b = true
int i = true;    // int(true) is converted to 1, so i = 1
int m = b + i;   // m = 1 + 1 = 2
```

### 1.3.5 The Void Type

A type that has no type at all is denoted by *void*. It is syntactically a fundamental type, but can be used only as part of a more complicated type. It is used either to specify that a function does not return a value or

as the base type for pointers to objects of unknown type. These points are explained later. See, for example, §3.8 and Exercise 3.14.24.

## 1.4 Numeric Limits

Machine-dependent aspects of a C++ implementation can be found in the standard library `<limits>`. For example, the function `numeric_limits<double>::max()` gives the largest *double* and function `numeric_limits<int>::min()` gives the smallest *int* that can be represented on a given computer. The library `<limits>` defines `numeric_limits<T>` as a template class (see Chapter 7) that has a type parameter *T*, where *T* can be *float*, *double*, *long double*, *int*, *short int*, *long int*, *char*, and *unsigned* integers. When *T* is, for example, *float*, then `numeric_limits<float>` gives implementation-dependent numbers for *float*. The following program prints out information on *float*.

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

main () {
    cout << "largest float = "
         << numeric_limits<float>::max() << '\n';
    cout << "smallest float = "
         << numeric_limits<float>::min() << '\n';
    cout << "min exponent in binary = "
         << numeric_limits<float>::min_exponent << '\n';
    cout << "min exponent in decimal = "
         << numeric_limits<float>::min_exponent10 << '\n';
    cout << "max exponent in binary = "
         << numeric_limits<float>::max_exponent << '\n';
    cout << "max exponent in decimal = "
         << numeric_limits<float>::max_exponent10 << '\n';
    cout << "# of binary digits in mantissa: "
         << numeric_limits<float>::digits << '\n';
    cout << "# of decimal digits in mantissa: "
         << numeric_limits<float>::digits10 << '\n';
    cout << "base of exponent in float: "
         << numeric_limits<float>::radix << '\n';
    cout << "infinity in float: "
         << numeric_limits<float>::infinity() << '\n';
    cout << "float epsilon = "
         << numeric_limits<float>::epsilon() << '\n';
    cout << "float rounding error = "
```

```

        << numeric_limits<float>::round_error() << '\n';
    cout << "float rounding style = "
        << numeric_limits<float>::round_style << '\n';
}

```

Similarly, implementation-dependent information on *double* and *long double* may be obtained as

```

main() {
    double smallestDouble = numeric_limits<double>::min();
    double doubleEps = numeric_limits<double>::epsilon();
    long double largestLongDouble =
        numeric_limits<long double>::max();
    long double longDoubleEpsilon =
        numeric_limits<long double>::epsilon();
}

```

and information on *char* may be obtained as

```

main() {
    cout << "number of digits in char: "
        << numeric_limits<char>::digits << '\n';
    cout << "char is signed or not: "
        << numeric_limits<char>::is_signed << '\n';
    cout << "smallest char: "
        << numeric_limits<char>::min() << '\n';
    cout << "biggest char: "
        << numeric_limits<char>::max() << '\n';
    cout << "is char an integral type: "
        << numeric_limits<char>::is_integer << '\n';
}

```

Some explanations are given now to the terms *epsilon* and *mantissa*. A machine *epsilon* is the smallest positive floating point number such that  $1 + \textit{epsilon} \neq 1$ . That is, any positive number smaller than it will be treated as zero when added to 1 in the computer. It is also called the *unit roundoff error*. When a floating point number  $x$  is represented in binary as  $x = \pm q \times 2^m$ , where  $q = 0.q_1q_2 \cdots q_n$  with  $q_1 = 1$  and  $q_i = 0$  or  $1$  for  $i = 2, 3, \dots, n$ , and  $m$  is an integer, then  $q$  is called the *mantissa* of  $x$ ,  $m$  is called the *exponent* of  $x$ , and  $n$  is the number of bits in the mantissa. Due to finite precision in computer representation, not all numbers, even within the range of the computer, can be represented exactly. A number that can be represented exactly on a computer is called a *machine number*. When a computer can not represent a number exactly, rounding, chopping, overflow, or underflow may occur.

On a hypothetical computer called *Marc-32* (see [CK99, KC96]; a typical computer should be very similar to it if not exactly the same), one machine



word (32 bits) is used to represent a single precision floating point number:

$$x = \pm 1.b_1b_2 \cdots b_{23} \times 2^m.$$

The leftmost bit in the machine word is the sign bit (0 for nonnegative and 1 for negative), the next 8 bits are for the exponent  $m$ , and the rightmost 23 bits for the fractional part ( $b_1, b_2, \dots$ , and  $b_{23}$ ) of the mantissa. Notice that, to save one bit of memory, the leading bit 1 in the fractional part is not stored. It is usually called the *hidden bit*. The exponent  $m$  takes values in the closed interval  $[-127, 128]$ . However,  $m = -127$  is reserved for  $\pm 0$ , and  $m = 128$  for  $\pm\infty$  (if  $b_1 = b_2 = \cdots = b_{23} = 1$ ) and *NaN* (otherwise). Thus the exponent of a nonzero machine number must be in the range  $-126 \leq m \leq 127$ . On such a machine, the single precision machine *epsilon* is then  $2^{-23} \approx 1.2 \times 10^{-7}$ , the smallest positive machine number is  $2^{-126} \approx 1.2 \times 10^{-38}$ , and the largest machine number is  $(2 - 2^{-23}) \times 2^{127} \approx 3.4 \times 10^{38}$  (corresponding to  $b_i = 1$  for  $i = 1, 2, \dots, 23$ , and  $m = 127$ ). In double precision, there are 52 bits allocated for the fractional part of the mantissa and 11 bits for the exponent, which results in the machine *epsilon* to be  $2^{-52} \approx 2.22 \times 10^{-16}$ , the smallest positive machine number  $2^{-1022} \approx 2.25 \times 10^{-308}$ , and the largest machine number  $(2 - 2^{-52}) \times 2^{1023} \approx 1.798 \times 10^{308}$ .

It can be shown (see [CK99, KC96]) that for any nonzero real number  $x$  and its floating point machine representation  $\bar{x}$  (assuming  $x$  is within the range of the computer and thus no overflow or underflow occurs), there holds

$$\left| \frac{x - \bar{x}}{x} \right| \leq \epsilon,$$

in the case of chopping, and

$$\left| \frac{x - \bar{x}}{x} \right| \leq \frac{1}{2}\epsilon,$$

in the case of rounding to the nearest machine number. That is, the relative roundoff error in representing a real number in the range of a computer with a particular precision is no larger than *epsilon* in that precision.

The machine *epsilon* and number of bits in binary (or equivalent digits in decimal) for representing the mantissa of a floating point number on a particular computer are given in the template class `numeric_limits<T>` for a floating point type  $T$ .

In addition, C library `<limits.h>` has macros (see §3.1 for some rules and examples for defining macros) such as those listed in Table 1.1, and C library `<float.h>` has macros such as those listed in Table 1.2. These macros are better avoided. However, the C++ library `<limits>` may not be available on early (nonstandard) compilers. In this case, numeric limits can be obtained as in the following program.

<i>INT_MAX</i>	largest <i>int</i>
<i>INT_MIN</i>	smallest <i>int</i>
<i>LONG_MAX</i>	largest <i>long int</i>
<i>LONG_MIN</i>	smallest <i>long int</i>
<i>ULONG_MAX</i>	largest <i>unsigned long int</i>
<i>UINT_MAX</i>	largest <i>unsigned int</i>
<i>SHRT_MAX</i>	largest <i>short int</i>
<i>USHRT_MAX</i>	largest <i>unsigned short int</i>
<i>SCHAR_MIN</i>	smallest <i>signed char</i>
<i>UCHAR_MAX</i>	largest <i>unsigned char</i>
<i>CHAR_MAX</i>	largest <i>char</i>
<i>CHAR_MIN</i>	smallest <i>char</i>
<i>WORD_BIT</i>	number of bits in one word
<i>CHAR_BIT</i>	number of bits in <i>char</i>

TABLE 1.1. Limits of integral numbers from C library &lt;limits.h&gt;.

<i>DBL_MAX</i>	largest <i>double</i>
<i>DBL_MIN</i>	smallest <i>double</i>
<i>DBL_EPSILON</i>	<i>double</i> epsilon
<i>DBL_MANT_DIG</i>	number of binary bits in mantissa
<i>DBL_DIG</i>	number of decimal digits in mantissa
<i>DBL_MAX_10_EXP</i>	largest exponent
<i>LDBL_MAX</i>	largest <i>long double</i>
<i>LDBL_MIN</i>	smallest <i>long double</i>
<i>LDBL_EPSILON</i>	<i>long double</i> epsilon
<i>LDBL_MANT_DIG</i>	number of binary bits in mantissa
<i>LDBL_DIG</i>	number of decimal digits in mantissa
<i>LDBL_MAX_10_EXP</i>	largest exponent
<i>FLT_MAX</i>	largest <i>float</i>
<i>FLT_MIN</i>	smallest <i>float</i>
<i>FLT_EPSILON</i>	<i>float</i> epsilon
<i>FLT_MANT_DIG</i>	number of binary bits in mantissa
<i>FLT_DIG</i>	number of decimal digits in mantissa
<i>FLT_MAX_10_EXP</i>	largest exponent

TABLE 1.2. Limits of floating point numbers from C library &lt;float.h&gt;.

```
#include <limits.h>
#include <float.h>

main() {
    int i = INT_MIN;           // smallest int
    long j = LONG_MAX;         // largest long int
    double x = DBL_MAX;        // biggest double
    long double y = LDBL_MAX;   // biggest long double
    float z = FLT_MAX;          // biggest float
    double epsdbl = DBL_EPSILON; // double epsilon
    float epsflt = FLT_EPSILON;  // float epsilon
    long double epsldb = LDBL_EPSILON; // long double epsilon
}
```

For headers ending with the *.h* suffix such as *float.h* and *math.h*, the include directive

```
#include <float.h>
```

automatically gives access to declarations in *float.h*, and the statement

```
using namespace std;
```

is not necessary.

Note that the C++ library *<limits>* is different from the C library *<limits.h>*, which C++ inherited from C. All C libraries can be called from a C++ program provided that their headers are appropriately included. See §4.2 for more details on all C and C++ standard header files.

## 1.5 Identifiers and Keywords

### 1.5.1 Identifiers

In our first program, we have used variables such as *sum* and *m* to hold values of different data types. The name of a variable must be a valid identifier. An identifier in C++ is a sequence of letters, digits, and the underscore character `_`. A letter or underscore must be the first character of an identifier. Upper and lower case letters are treated as being distinct. Some identifiers are:

```
double sum = 0;           // identifier sum suggests a summation
double product;           // identifier product suggests multiply
bool flag;                // flag certain condition (true or false)

int Count;
int count;                // different from Count
int this_unusually_long_identifier; // legal, but too long
```

<b>and</b>	<b>and_eq</b>	<b>asm</b>	<b>auto</b>
<b>bitand</b>	<b>bitor</b>	<b>bool</b>	<b>break</b>
<b>case</b>	<b>catch</b>	<b>char</b>	<b>class</b>
<b>compl</b>	<b>const</b>	<b>const_cast</b>	<b>continue</b>
<b>default</b>	<b>delete</b>	<b>do</b>	<b>double</b>
<b>dynamic_cast</b>	<b>else</b>	<b>enum</b>	<b>explicit</b>
<b>export</b>	<b>extern</b>	<b>false</b>	<b>float</b>
<b>for</b>	<b>friend</b>	<b>goto</b>	<b>if</b>
<b>inline</b>	<b>int</b>	<b>long</b>	<b>mutable</b>
<b>namespace</b>	<b>new</b>	<b>not</b>	<b>not_eq</b>
<b>operator</b>	<b>or</b>	<b>or_eq</b>	<b>private</b>
<b>protected</b>	<b>public</b>	<b>register</b>	<b>reinterpret_cast</b>
<b>return</b>	<b>short</b>	<b>signed</b>	<b>sizeof</b>
<b>static</b>	<b>static_cast</b>	<b>struct</b>	<b>switch</b>
<b>template</b>	<b>this</b>	<b>throw</b>	<b>true</b>
<b>try</b>	<b>typedef</b>	<b>typeid</b>	<b>typename</b>
<b>union</b>	<b>unsigned</b>	<b>using</b>	<b>virtual</b>
<b>void</b>	<b>volatile</b>	<b>wchar_t</b>	<b>while</b>
<b>xor</b>	<b>xor_eq</b>		

TABLE 1.3. All C++ keywords.

```
int _aAbBcCdD;                                // legal, but no meaning
```

It is good programming practice to choose identifiers that have mnemonic significance so that they contribute to the readability and documentation of a program. Confusing identifiers should be avoided. For example, *ll*, *lo*, and *lO* are different and valid identifiers but are hard to read, and identifiers *Count* and *count* can be easily misunderstood.

### 1.5.2 Keywords

Keywords such as *int*, *double*, and *for* are explicitly reserved identifiers that have a strict meaning in C++. They can not be redefined or used in other contexts. For example, a programmer can not declare a variable with the name *double*. A keyword is also called a reserved word. A complete list of C++ keywords is in Table 1.3.

Note that *cin* and *cout*, for example, are not keywords. They are part of the input and output (I/O) library `<iostream>`.

## 1.6 Exercises

- 1.6.1. Modify the program in §1.1 to compute the sum of the squares of all integers between two given integers. That is, find the sum  $n^2 + (n+1)^2 + \dots + m^2$  for two given integers  $n$  and  $m$  with  $n < m$ . In the sample program in §1.1, the variable *sum* should be declared as *double* or *long double* in order to handle large values of  $n$  and  $m$ . In this exercise, try to compute *sum* in two ways: as a *long double* and as an *int*, and compare the results. On a computer with 4 bytes for storing *int*, the second way calculates the sum  $1^2 + 2^2 + 3^2 + \dots + 5000^2$  as  $-1270505460$ . Why could a negative number as the output be possible?
- 1.6.2. Modify the program in §1.1 to multiply all integers between two given small positive (e.g., less than or equal to 12) integers. When one of them is 1 and the other is a positive integer  $n$ , the program should find the factorial  $n! = 1 * 2 * 3 * \dots * (n-1) * n$ . The program to find  $n!$  may be written as

```
#include <iostream>
using namespace std;
main() {
    int n;
    cout << "Enter a positive integer: \n";
    cin >> n;

    int fac = 1;
    for (int i = 2; i <= n; i++) fac *= i; // fac = fac*i;
    cout << n << "! is: " << fac << '\n';
}
```

Except for possible efficiency difference, the statement  $fac * = i$  is equivalent to  $fac = fac * i$ .

This simple program can only compute  $n!$  correctly for  $n = 1$  up to  $n = 12$  on a computer with  $sizeof(int) = 4$ . When  $n = 13$ , such a computer may calculate  $13!$  as 1932053504 while the correct value is  $13! = 6227020800$ . It may also compute  $20!$  as  $-2102132736$  (why negative?) and  $40!$  as 0 (why just 0?). A computer can produce garbage very easily. Test what your computer gives you for  $13!$ ,  $20!$ , and  $40!$ . A user should check the correctness of computer outputs by all means. Sometimes outputs such as the erroneous result for  $13!$  above can be very hard to check. Outline a procedure to determine the correctness of your computer output of  $13!$ , assuming you do not know the correct value.

In Exercise 3.14.21, a technique is introduced that can compute  $n!$  correctly for much larger  $n$ , for example,  $n = 3000$  or larger. Notice that the number  $3000!$  has 9131 digits that would overflow as an integer on any current computer.

1.6.3. If one wishes to compute the following summation

$$\sin(1.1) + \sin(1.3) + \sin(1.5) + \cdots + \sin(9.9),$$

the program in §1.1 can be modified to do so:

```
#include <iostream>           // input/output library
#include <math.h>              // math library for sin

main() {
    double sum = 0;           // sum initialized to 0

    for (double d = 1.1; d <= 9.9; d += 0.2) sum += sin(d);
    cout << "The sum is: " << sum << '\n';
}
```

This *for* loop declares  $d$  to be a variable of double precision with initial value 1.1, and executes the statement  $sum += \sin(d)$  for  $d$  changing from 1.1 up to 9.9 with increment 0.2 each time. Compile and run this program, and modify it to compute

$$e^{1.1} + e^{1.2} + e^{1.3} + \cdots + e^{15.5},$$

where  $e$  is the base of the natural logarithm. See §3.11 for a complete list of mathematical functions in the library `<math.h>`.

- 1.6.4. Write a program that outputs the largest integer, smallest integer, and the number of bytes for storing an integer on your computer. Also do this with *short int*, *int*, and *long int*.
- 1.6.5. Write a program that outputs the integer values corresponding to characters `A`, `9`, `a`, `{`, `$`, `\n` (new line), `\t` (horizontal tab), `\0` (null character), `\\` (backslash), `\r` (carriage return), `\"` (double quote), `\b` (backspace), `\f` (formfeed), `\'` (single quote), `\v` (vertical tab), `\?` (question mark), and `\a` (alert) on your computer.
- 1.6.6. Write a program that outputs exactly the sentences:  
 He said: "I can output double quotation marks."  
 She asked: "Do you know how to output the newline character `\n`?"  
 Notice that the double quotation mark, newline, and question mark are special characters in C++. See Exercise 1.6.5

- 1.6.7. A backslash symbol `\` at the end of a line is supposed to continue it to the next line. Test the following program to see its effect.

```
#include <iostream>
using namespace std;
main() {
    cout << "I am outputting a string that stands on \
three lines to test the effect of a continuation line \
using a backslash\n";
}
```

- 1.6.8. Write a program that outputs the largest and smallest numbers, *epsilon*, the number of decimal digits used to store the mantissa, and the largest exponent for double precision floating point numbers on your computer. Repeat this for *float* and *long double* as well.

- 1.6.9. Compile and run the program

```
#include <iostream>
using namespace std;
main() {
    long g = 12345678912345;
    short h = g;           // beware of integer overflow
    int i = g - h;
    cout << "long int g = " << g << '\n';
    cout << "short int h = " << h << '\n';
    cout << "their difference g - h = " << g - h << '\n';
}
```

on your computer. Does your compiler warn you about *integer overflow*? It may not always do so. Does your computer give  $g - h = 0$ ? Beware of overflow and truncation in converting a larger type to a smaller one.

- 1.6.10. Calculate the value of  $\pi$  on your computer following the steps in §1.3.3 in single, double, and extended double precisions. How many digits of accuracy does your computer give in each of the three floating point types?
- 1.6.11. What is the value of  $i$  in the following statement?

```
int i = 3.8 + 3.8;
```

Is it 6, 7, or 8? Type conversion may lead to loss of accuracy. Does your compiler warn you about this? It may not always do so.

1.6.12. What do you think the following program will output?

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

int main() {
    double x = DBL_MAX;           // biggest double
    double epsilon = DBL_EPSILON; // double epsilon
    double zero = 0.0;
    double y = 100.2;

    double z0 = x + x;
    double z1 = x * 2;
    double z2 = epsilon/9;
    double z3 = y/zero;
    double z4 = zero/zero;
    double z5 = z3 - z3;
    double z6 = x + y;

    cout << "outputting results:\n";
    cout << z0 << '\n';
    cout << z1 << '\n';
    cout << z2 << '\n';
    cout << z3 << '\n';
    cout << z4 << '\n';
    cout << z5 << '\n';
    cout << z6 << '\n';
    cout << 1 + z2 << '\n';

}
```

Run the program on your computer to check the results. You may see Infinity, NaN, and other unusual outputs.



# 2

## Expressions and Statements

The code segment

```

int n;
int m = 10;
n = m + 5;

```

contains three *statements*. The first one is a *declaration* (it declares  $n$  to be an integer; i.e., it informs the compiler that  $n$  is a variable of type *int*, and the compiler will allocate appropriate memory space for it), the second one is a declaration together with an initialization (it declares  $m$  to be an integer and initializes it to 10), and the third statement is an *assignment* (it assigns the value of  $m$  plus 5 to  $n$ ) that contains an *expression*  $m + 5$  with operands  $m$  and 5 and operator  $+$ . A *declaration* is a statement that introduces a name associated with a type into the program. All statements must end with a semicolon. In this chapter, various expressions and statements are discussed, after starting with scopes and storage classifications. The chapter finishes by providing a sample program on calculating Fibonacci numbers.

### 2.1 Scopes and Storage Classifications

A *scope* is part of a program text consisting of statements enclosed by a pair of braces, except for the *global scope* that is outside all braces. Variables defined in the global scope are called *global variables*, and those defined in local scopes are called *local variables*. This section deals with scopes and different kinds of variables.

### 2.1.1 Local and Global Variables

Unlike C and FORTRAN, a declaration is a statement that can occur anywhere another statement can, not necessarily at the beginning of a program. If it occurs within a *block* enclosed by a pair of braces, it then declares a *local variable* to the block. This block introduces a *scope*. Variables local to a scope are inaccessible outside the block defining the scope. For example,

```
int n;
{
    // this pair of braces introduces a scope
    int m = 10;    // m is accessible only within this scope
}
n = m;            // error, m is out of scope
int m = 20;       // OK, it declares a new variable m
```

The local variable  $m = 10$  is not accessible outside the scope in which it is defined. Thus the assignment  $n = m$  is illegal and the compiler treats  $m$  as not having been declared. However, it is legal to define another variable  $m = 20$  after that scope. Variables can have the same name if they are in different scopes.

A variable declared before a block can be accessed after and within the block, and within all inner blocks, provided it is not hidden by another variable declared within such a block and having the same name. A variable declared outside all blocks is said to be a *global variable*. When a local variable and a global variable have the same name, the scope resolution operator `::` can be used to access the global variable. For example,

```
int x = 10;        // this x is global (outside all blocks)
main() {
    int x = 25;     // local variable x, it hides global x
    int y = ::x;    // y = 10, ::x refers to global variable x
    {
        int z = x;  // z = 25; x is taken from previous scope
        int x = 38; // it hides local variable x = 25 above
        int t = ::x; // t = 10, ::x refers to global variable x
        t = x;      // t = 38, x is treated as local variable
    }
    int z = x;      // z = 25, z has same scope as y
}
```

There is no way to access a hidden local variable. For example, the local variable  $x = 25$  is hidden and inaccessible from the scope where integer  $t$  is defined. Note that the local variables  $x = 25$  and  $y$  have the same scope as  $z$ , the local variable introduced in the last statement of the program.

An *automatic variable* is created at the time of its declaration and destroyed at the end of the block in which it is declared, as the variables  $y$ ,  $t$ , and  $z$  above. When the block is entered, the system sets aside memory

for automatic variables. When the block is exited, the system no longer reserves the memory for these variables and thus their values are lost. A local variable is *automatic* by default. It can also be declared explicitly by the keyword *auto*. For example, the declaration `int x = 38;` above is equivalent to: `auto int x = 38;`.

### 2.1.2 External and Register Variables

A variable defined in one file can be accessed in another file by declaring it to be external using the keyword *extern*:

```
extern double x;    // x can be accessed across files
```

This declaration tells the system that variable *x* is defined externally, perhaps in another file (see §4.3.2 for more details). However, it does not allocate space for the external variable *x* and simply means that *x* is accessible in all files that have the statement `extern double x;`. It must be declared as usual in one and only one file:

```
double x;           // declare it as usual in one of the files
```

It is this statement that creates an object and allocates space for *x*. (See §4.4.2 on how to compile a program consisting of more than one file.)

**External variables never disappear; they exist throughout the execution life of the program.** To improve the modularity and readability of a program, the use of external variables should be kept to the minimum. In particular, a function (see §3.8) should not change an external variable within its body rather than through its parameter list or return value. An external variable can sometimes be localized:

```
{
    // some code
    {
        extern double x;    // x is restricted to this scope only
        double y = x;
    }
}
```

The declaration of a *register variable* indicates that it is stored in high speed registers. Since resource limitations and semantic constraints sometimes make this impossible, this storage classification defaults to *automatic* whenever the compiler can not allocate an appropriate physical register. The use of register variables is an attempt to improve execution speed and the programmer may declare a few most frequently used variables as *register*. For example,

```
for (register int i = 0; i < 10000; i++) {
    // ... do something
```

```
}

```

However, modern compilers can normally do a better job than a programmer as to which variable should be assigned to a register.

Thus every variable has two attributes: *type* and *storage classification*.

Three storage classifications have been talked about in this section. They are *automatic*, *external*, and *register*, which are represented by keywords *auto*, *extern*, and *register*. How these variables are stored distinguishes them from each other. The use of C-style *static* storage classification, defining a variable local to only one file, is discouraged in C++, since its use is subtle and the keyword *static* is overused in C++. However, C++ provides *namespaces* for defining variables local to a file; see §4.1.4.

## 2.2 Expressions

If we define `int x = 25; int y = 10;` then `x - y` is called an *expression* having operands `x` and `y` and operator `-`, the value of which is 15. In this case, `-` is called a *binary operator* since it takes two operands. It can also be used as a *unary operator* as in `x = -5`, which takes only one operand. Different expressions are introduced in this section.

### 2.2.1 Arithmetic Expressions

*Arithmetic expressions* contain arithmetic operations for addition, subtraction, multiplication, and division, whose corresponding operators are `+`, `-`, `*`, and `/`, respectively. `+` and `-` are also unary operators. These operators are defined for all integer and floating point types and there are appropriate type conversions between different types. The *remainder* of one integer divided by another is given by the *modulo operator* `%`. Note that the division of one integer by another has an integer value in C++, where the remainder (zero or not) is ignored. For example,

```
double m = 13/4;           // m = 3, remainder is ignored
double n = 3/4;            // n = 0, it just gives quotient
double x = double(3)/4;    // x = 0.75, 3 is converted to 3.0
double y = 3.0/4;          // y = 0.75
double z = 3/4.0;          // z = 0.75
int t = 13%4;              // t = 1, remainder of 13/4 is 1
```

Left to right associativity holds for arithmetic expressions. Multiplication and division have higher precedence than addition and subtraction, and parentheses override this precedence as in mathematical expressions. For example,

```
m = 1 + 2 + 3*4;           // m = 3 + 12 = 15
```

```
m = 1 + (2 + 3)*4;           // m = 1 + 5*4 = 21
m = exp1 + (exp2 - exp3);
```

In general, the following two statements are equivalent,

```
m = exp1 + exp2 + exp3;
m = (exp1 + exp2) + exp3;    // left to right associativity
```

although it is not guaranteed that *exp1* or *exp2* is evaluated before the expression *exp3* is evaluated. That is, associativity and precedence only group subexpressions together, but do not imply which subexpression is evaluated first in time.

### 2.2.2 Relational Expressions

*Relational operators* are  $>$  (bigger than),  $<$  (smaller than),  $>=$  (bigger than or equal to),  $<=$  (smaller than or equal to),  $==$  (equal), and  $!=$  (not equal). The notation  $!$  is the negation operator. The result of evaluating these operators is of type *bool*; it is either *true* or *false*. For example,

```
int x = 5;
bool b = (x < 6);           // b = true, since x is less than 6
bool c = (x == 0);          // c = false, since x is not 0
if (b != c) x = 17;         // if b is not equal to c, let x = 17
if (b) x = 19;              // if b is true, then do x = 19
if (!b) x = 221;            // if b is false, then do x = 221
```

In general, a zero quantity is *false* and a nonzero quantity is *true*. Care must be taken when testing for equality of two floating point numbers owing to finite precision in computer representation and roundoff errors; see Exercise 2.5.10. Instead of doing so directly, one usually checks if the absolute value of their difference is less than a small number. Notice that  $m = n$  is an assignment while  $m == n$  is a relational expression.

Mathematical and C++ expressions may sometimes have surprisingly different interpretations. For example, the mathematical expression

$$3 < m < 9$$

is true for  $m = 7$  and is false for  $m = 20$ . However, consider the C++ code:

```
int m = 7;
bool b = 3 < m < 9;         // b = true

m = 20;
bool c = 3 < m < 9;         // c = true
```

Due to the left to right associativity for relational operators, the C++ expression  $3 < m < 9$  is equivalent to  $(3 < m) < 9$ . Thus the C++

expression  $3 < m < 9$  always evaluates to *true* because  $3 < m$  is evaluated to either 1 (true) or 0 (false).

Another example is that the mathematical expression

$$x < x + y$$

is always true for any positive real number  $y$  and any real number  $x$ , while the C++ expression

```
x < x + y;
```

evaluates to *false* for some large value of  $x$  (e.g.,  $x = 5\text{e}+31$  in 12-byte long double precision) and small value of  $y$  (e.g.,  $y = 0.1$ ) since, in this case,  $x$  and  $x + y$  are treated as being equal (due to finite precision and roundoff error) on the computer. Similarly, the expression  $x + y > x - y$  is always true for any positive numbers  $x$  and  $y$  in mathematics, but not in C++.

### 2.2.3 Logical Expressions

*Logical operators* are `&&` (*and*) and `||` (*or*). A logical expression has value of type *bool*. If *exp1* and *exp2* are two expressions, then *exp1* `&&` *exp2* has value *true* if both *exp1* and *exp2* are true and has value *false* otherwise; and *exp1* `||` *exp2* has value *true* if at least one of *exp1* and *exp2* is true and has value *false* otherwise. This can be more easily illustrated by using the so-called truth table.

<i>true</i>	<code>&amp;&amp;</code>	<i>true</i>	<code>= true</code>	<i>true</i>	<code>  </code>	<i>true</i>	<code>= true</code>
<i>true</i>	<code>&amp;&amp;</code>	<i>false</i>	<code>= false</code>	<i>true</i>	<code>  </code>	<i>false</i>	<code>= true</code>
<i>false</i>	<code>&amp;&amp;</code>	<i>true</i>	<code>= false</code>	<i>false</i>	<code>  </code>	<i>true</i>	<code>= true</code>
<i>false</i>	<code>&amp;&amp;</code>	<i>false</i>	<code>= false</code>	<i>false</i>	<code>  </code>	<i>false</i>	<code>= false</code>

Some examples of logical expressions are:

```
bool b = true, c = false;
bool d = b && c;           // d = false
bool e = b || c;          // e = true
bool f = (e == false) && c; // f = false
bool g = (d == true) || c; // g = false
bool h = (d = true) || c;  // h = true
```

Note that in the last statement above,  $d$  is assigned to be *true* in  $d = \text{true}$  and thus  $(d = \text{true}) \parallel c = \text{true} \parallel \text{false} = \text{true}$ . Again,  $d == \text{true}$  is a relational expression while  $d = \text{true}$  is an assignment.

The logical expression

```
i >= 0 && i < n
```

tests if  $i$  is bigger than or equal to 0 and less than  $n$ . If  $i$  is less than 0, then the value of the logical expression is *false* no matter what the value of  $i < n$  is, and the subexpression  $i < n$  is skipped without evaluation. This is the so-called *short-circuit evaluation* and should lead to a faster evaluation of such logical expressions. It also applies to the logical *or* operator `||`. It is guaranteed that evaluation is from left to right and the process stops as soon as the outcome *true* or *false* is known. That is, in this example, the subexpression on the left  $i \geq 0$  is evaluated first and then the subexpression  $i < n$  may or may not be evaluated depending on the value of the left subexpression. This can be more easily seen from the following examples.

```
int i, j = 2;
bool k = ((i = 0) && (j = 3));    // i = 0, k = false, j = 2
bool m = ((i = 4) || (j = 5));    // i = 4, m = true, j = 2
bool n = ((i = 0) || (j = 6));    // i = 0, j = 6, n = true
```

In the second statement above,  $i$  is assigned to be 0 and the left subexpression  $(i = 0)$  is converted to *false*. Thus  $k = \text{false}$  no matter what value the right subexpression  $(j = 3)$  has, and the right subexpression  $(j = 3)$  is skipped without evaluation. In the third statement,  $i$  is assigned to be 4 and  $(i = 4)$  is converted to *true*, and the right subexpression  $(j = 5)$  is skipped.

Notice that this kind of left to right evaluation is not true for arithmetic expressions. For example, in  $m = \text{exp1} + \text{exp2}$ ; it is not guaranteed that  $\text{exp1}$  is evaluated before  $\text{exp2}$ .

Referring to an example in §2.2.2, the correct way of translating the mathematical expression  $3 < m < 9$  into C++ is

```
(3 < m) && (m < 9)
```

Since operator `<` has higher precedence than operator `&&`, the two pairs of parentheses above may be omitted.

It is defined that the operator `&&` has higher precedence than `||`. For example, the following two statements are equivalent,

```
bool m = i || j && k;
bool m = i || (j && k);
```

although  $x = i || j || k$  is equivalent to  $x = (i || j) || k$  (left to right associativity). Parentheses can override precedence and make code more readable.

A summary on precedence for all C++ operators is given in §3.10.

#### 2.2.4 Bitwise Expressions

*Bitwise operators* are `&` (and), `|` (or), `^` (exclusive or), `~` (complement), `<<` (left shift), and `>>` (right shift), which can be applied to objects of

integral type (*bool*, *char*, *short*, *int*, *long*, and their *unsigned* counterparts). In these expressions, operands are first implicitly represented in binary (string of binary bits) and then operation is performed on them bit by bit. Bitwise operations can be machine-dependent and may not be portable, since, for example, different machines may represent integers differently and use different bytes for storing them.

Binary bitwise operations  $\&$ ,  $|$ , and  $\wedge$  are performed by comparing their operands bit by bit. If two bits are both 1, then  $\&$  and  $|$  give 1, and  $\wedge$  gives 0. If two bits are both 0, then  $\&$ ,  $\wedge$ , and  $|$  all give 0. If one bit is 0 and the other is 1, then  $|$  and  $\wedge$  give 1, and  $\&$  gives 0. This can be more easily illustrated by the following table.

$1 \& 1 = 1$	$1 \& 0 = 0$	$0 \& 1 = 0$	$0 \& 0 = 0$
$1   1 = 1$	$1   0 = 1$	$0   1 = 1$	$0   0 = 0$
$1 \wedge 1 = 0$	$1 \wedge 0 = 1$	$0 \wedge 1 = 1$	$0 \wedge 0 = 0$

For example, 13 and 7 are in binary  $0 \cdots 01101$  and  $0 \cdots 00111$ , respectively. Then

```
int a = 13 & 7;           // & is performed bit by bit
```

defines  $a$  to have a binary representation  $0 \cdots 00101$ , which is obtained by applying the bitwise  $\&$  operator to corresponding bits of the binary representations of 13 and 7. This gives a decimal number  $2^2 + 2^0 = 5$ . Similarly, the statements

```
int b = 13 ^ 7;           // b = 10
int c = 13 | 7;           // c = 15
```

define  $b$  to have a binary representation  $0 \cdots 01010$ , which is  $2^3 + 2 = 10$ , and  $c$  a binary representation  $0 \cdots 01111$ , which is  $2^3 + 2^2 + 2^1 + 2^0 = 15$ .

If  $n$  and  $s$  are integers, then  $n \ll s$  shifts the pattern of bits of  $n$ ,  $s$  places to the left, putting  $s$  zeros in the rightmost places. On the other hand,  $n \gg s$  shifts the pattern  $s$  places to the right, putting  $s$  zeros in the leftmost places for *unsigned* integers. (For *signed* integers,  $s$  zeros or  $s$  ones may be shifted in depending on the machine.) For example, for an *unsigned int*  $n = 16$ , consider the statements

```
int m = n << 2;           // m = 64
int k = n >> 2;           // k = 4
```

Since  $16 = 2^4$  has a binary representation  $0 \cdots 010000$ , left shifting its bits by 2 positions leads to a binary pattern  $0 \cdots 01000000$ , which is  $2^6 = 64$ , and right shifting its bits by 2 positions leads to  $0 \cdots 0100$ , which is  $2^2 = 4$ . Note that  $\ll$  and  $\gg$  are also input and output operators and they can be easily distinguished from the context.

The bitwise complement operator  $\sim$  is unary and changes every 0 into 1 and every 1 into 0 in the binary representation of its operand. For example,



the integer 0 has all bits equal to 0 in its binary representation and  $\sim 0$  has all bits equal to 1.

The value of  $\sim n$  is called *one's complement* of  $n$ . *Two's complement* of an integer  $n$  is the value of adding 1 to its one's complement. On a two's complement machine, a nonnegative integer  $n$  is represented by its bit representation and a negative integer  $-n$  is represented by the two's complement of  $n$ . For example, 7 is represented by the binary string  $0000 \cdots 0111$ , and  $-7$  is represented by  $1111 \cdots 1001$ , which is obtained from adding 1 to  $1111 \cdots 1000$ , the complement of 7.

It can be checked that the two's complement of a negative integer  $-n$  has value  $n$ . For example,  $-7$  is represented by  $1111 \cdots 1001$ . Its two's complement is  $0000 \cdots 0111$ , which is 7. In two's complement, the integer 0 has all bits off and  $-1$  has all bits on. Notice that the leftmost bit represents the sign of the integer: 1 for negative and 0 for nonnegative. On a two's complement machine, the hardware that does addition can also do subtraction. For example, the operation  $a - b$  is the same as  $a + (-b)$ , where  $-b$  is obtained by taking  $b$ 's two's complement.

In Exercise 2.5.15, a program is given to print out the bit representation of any integer on a computer.

### 2.2.5 Comma Expressions

The expression

```
i = 3, i + 2;
```

is called a *comma expression* with a comma used to separate subexpressions. Its value is the value of the last subexpression, and in this example its value is  $i + 2 = 5$ . A comma expression is guaranteed to be evaluated from left to right. Thus, the subexpression  $i = 3$  is evaluated first and then the subexpression  $i + 2$ . Consequently, the statement

```
j = (i = 3, i + 2);
```

gives  $j = 5$ .

## 2.3 Statements

A program consists of a collection of statements. Each statement must end with a semicolon. The null statement is simply a semicolon, which is used when the syntax requires a statement but no action is desired. In this section, we deal with various statements: declarations, assignments, compound statements, conditional statements, iteration statements (loops), jump statements, and selection statements. Later we encounter return statements and function calls (see Chapter 3).

### 2.3.1 Declarations and Initializations

The *declaration*

```
int n;
```

declares the variable  $n$  to be an integer and reserves for it `sizeof(int)` bytes of addressable memory, the content of which is in general undefined. The system may initialize  $n$  with a default value in some situations, but it is safe to always assume the value of  $n$  is undefined. To give it a meaningful value, an assignment such as

```
n = 10;
```

can be used. A variable can be declared and initialized in one statement such as

```
int n = 10;
```

This is called an *initialization*. All variables must be declared before they are used. One declaration statement can declare several variables of the same type, separated by commas, such as

```
double x, y, z;          // declare x, y, z in one declaration
```

It is equivalent to declaring them separately.

The particular block of memory reserved for a variable is sometimes called an *object*. For example, when we declare an integer variable  $n$ :

```
int n = 10;
```

a certain contiguous region of memory (`sizeof(int)` bytes) is allocated for it. This particular contiguous region of memory is called an *object*. The name of this object is  $n$ . There are two numbers associated with the object  $n$ : its value (the value stored in the block of memory), written as  $n$ , and its *location* or *address* of the object in memory, written as  $\&n$ . The symbol  $\&$  is called the *address-of* operator. It gives the address of an object. We can change the value of object  $n$  through the assignment:  $n = 55$ ; It simply stores the new value 55 at the same memory location. This can be checked by the following program.

```
main() {
    int n = 10;
    cout << "The value of variable n is: " << n << '\n';
    cout << "The location in memory of variable n is: " << &n;

    n = 55;
    cout << "The value of variable n is: " << n << '\n';
    cout << "The location in memory of variable n is: " << &n;
}
```

Locations of objects in memory are implemented using large integers. In §3.6, pointers are introduced as a data type to represent and manipulate locations of objects.

### 2.3.2 Assignments

If  $a, b, c$  have type *double*, then

```
a = b + c;
```

is an example of an assignment, which assigns the sum of values of objects  $b$  and  $c$  to the object named  $a$  at the address  $\&a$ . Here  $b + c$  can be called the *rvalue*, and  $a$  the *lvalue*. The term *lvalue* comes from “left-hand side value,” and *rvalue* “right-hand side value.” An *lvalue* is defined to be an expression that refers to an object. An *rvalue* is obtained by *reading* the content of an object and an *lvalue* is associated with *writing*. Therefore, a statement like  $a + b = c$  is meaningless and illegal, since  $a + b$  can not be used as an *lvalue*.

Also note that an initialization and an assignment are two distinct operations. An *initialization* allocates space in memory for a variable and initializes a value for the variable at its location and thus creates an object. An *assignment* does not create an object but simply assigns a value to an object already created before.

### 2.3.3 Compound Assignments

The assignment operator may be combined with any of the binary arithmetic or bitwise operators to form a *compound assignment* operator. If  $@$  is one of these operators, then  $x @ = y$  means  $x = x @ y$ , except for possible efficiency differences and side effects (see Exercises 2.5.6 and 3.14.12). For example,  $x /= 5$  means  $x = x / 5$  (division), and  $x \&= 5$  means  $x = x \& 5$  (bitwise and). Notice that there is no whitespace between the two operators and they form a single operator. The complete set of compound assignment operators are  $+=$ ,  $-=$ ,  $*=$ ,  $/=$ ,  $\%=$ ,  $\&=$ ,  $|=$ ,  $\hat{=}$ ,  $\ll=$ ,  $\gg=$ .

Compound assignment (also increment and decrement; see §2.3.4) operators have the potential to be implemented efficiently. For example, an intermediate value of  $x - 5$  is usually obtained and stored and then assigned to  $x$  in  $x = x - 5$ ; while in  $x -= 5$ ; such an intermediate process can be omitted and the value of  $x$  is just decremented by 5. They may also save some typing and be more readable as in the following two statements.

```
my_very_long_variable = my_very_long_variable - 5;  // messy
my_very_long_variable -= 5;                          // better
```

### 2.3.4 Increments and Decrements

When increasing or decreasing a variable by unity, more compact notation can be used. If  $n$  is a variable,  $n++$  and  $++n$  both increase the value of  $n$  by 1. In the postfix form  $n++$ , the original value of  $n$  is used in the expression in which it occurs before its value is incremented. In the prefix form  $++n$ , the value of  $n$  is first incremented by 1 and then the new value of  $n$  is used. A similar explanation holds for the *decrement operator*  $--$ . For example,

```
int i = 4, j = 4;    // i and j are both initialized to 4
i++;                // i = i+ 1. So i= 5
++j;                // j = j+ 1. So j= 5
int m = i++;        // first m=i, then i=i+1. So m=5, i=6
int n = ++j;        // first j=j+1, then n=j. So n=6, j=6
m = --i;            // first i=i-1, then m=i. So m=5, i=5
n = j--;            // first n=j, then j=j-1. So n=6, j=5
```

In the statement  $m = --i$ , the value of  $i$  is decremented by 1 first and the new value of  $i$  is assigned to  $m$ , while in the statement  $n = j--$ , the value of  $j$  is first assigned to  $n$  and then the value of  $j$  is decremented by 1.

Compound operators may sometimes cause side effects. For example, the statement

```
j = (i--) + (++i);    // unpredictable result
```

is a legal statement but its behavior is unpredictable and thus should not be used, since it is nondeterministic to evaluate the left operand  $i--$  or the right operand  $++i$  first. Suppose  $i = 5$ . Then evaluating the left operand  $i--$  before the right operand  $++i$  would give  $j = 10$ . Reversing the order of evaluation would give  $j = 12$ . See Exercises 2.5.6 and 3.14.12. Compound operators should be used in a straightforward way. Otherwise they can easily lead to less readable and error-prone code.

### 2.3.5 Compound Statements

A *compound statement* is a block of statements enclosed by a pair of braces. For example,

```
{
    int n = 10;
    n += 5;
}
```

is a compound statement, and it creates a new *scope*. Notice that no semicolon is needed after the right brace symbol. If there were a semicolon, it would be a null statement following the compound statement. Since

a compound statement is syntactically equivalent to a single statement, compound statements can be nested. That is, a compound statement can appear inside another compound statement. For example,

```
{
    int n = 10;
    n |= 5;           // n = n | 5 (bitwise or)
    {                // a nested compound statement
        double m = 20;
        m *= 3.14;
    }
}
```

### 2.3.6 Conditional Statements

#### *if-else* Statements

The simplest conditional statement is the *if*-statement, whose syntax is:

*if* (*condition*) *statement*

where *condition* is a logical expression having value *true* or *false*, and *statement* can be a single or compound statement. The *statement* is executed when the value of *condition* evaluates to *true* and is skipped otherwise. The *condition* must be enclosed in parentheses. For example,

```
if (n > 0) x *= 10;           // when n > 0, do x *= 10
if (n == 0) {                // when n is 0, do x *= 5
    x *= 5;
}
if (n) y *= 7;               // when n != 0, do y *= 7
```

Notice that an implicit conversion is made in the third *if*-statement; *y* \*= 7 is executed when *n* has a nonzero value (which is converted to *true*) and skipped when *n* has a zero value (which is converted to *false*).

The *if*-statement should be used with caution when the conditional expression requires the comparison for variables not of integral type. For example, if *x* and *y* are doubles, then the condition *if* (*x* == *y*) may be replaced by *if* (*fabs*(*x* - *y*) <= *Small*), where *Small* is a user-defined very small number and *fabs* is a math function that gives the absolute value of a floating point number. See Exercise 2.5.10.

The difference between *if* (*x* == *y*) and *if* (*x* = *y*) should also be noticed. Such small points can be hard to detect when they are buried in a program.

The *if*-statement can have an optional *else* part. Its syntax is

*if* (*condition*) *statement1*  
*else statement2*

where *condition* is a logical expression and *statement1* and *statement2* are single or compound statements. *statement1* is executed and *statement2* is skipped when the value of *condition* evaluates to *true*, and *statement2* is executed and *statement1* is skipped otherwise. For example,

```
if (x == 0) cout << "denominator is zero\n"; // warning msg
else y /= x;                               // do division
```

When *x* is not 0, a division by *x* is performed. Otherwise output a warning message.

The *if-else* statement can be nested. For example, in the statement

```
if (condition1) statement1
else if (condition2) statement2
else if (condition3) statement3
else statement4
```

*statement1* is executed if and only if *condition1* is *true*. If it is *true*, the rest is skipped; if it is *false*, *statement1* is skipped, and the value of *condition2* is checked. If *condition2* is *true*, then *statement2* is executed and *statement3* and *statement4* are skipped; if it is *false*, *statement2* is skipped and the value of *condition3* is checked. If *condition3* is *true*, then *statement3* is executed and *statement4* is skipped; if it is *false*, then *statement3* is skipped and *statement4* is executed. The compound *if-else* statement above can be equivalently written as

```
if (condition1) statement1
else {
    if (condition2) statement2
    else {
        if (condition3) statement3
        else statement4
    }
}
```

An *if* or *if-else* statement defines a new *scope* and declaration of a new variable is possible, even in the *condition* part. For example,

```
double x; // #include <stdlib.h>
if (int r = rand()) { // r is declared in condition part
    x /= r;           // if r is nonzero, do a division
} else {              // if r is zero, do an addition
    x += r;           // r is also within scope here
}
double y = r; // illegal, r is out of scope
```

The function *rand()*, declared in the standard header *<stdlib.h>*, generates a pseudorandom number between 0 and *RAND\_MAX*. The macro *RAND\_MAX* is also defined in *<stdlib.h>*. To generate a pseudorandom

number between 0 and  $n$ , using `int((double(rand()) / RAND_MAX) * n)` gives a more random result than simply `rand() % n`.

The *ternary conditional operator* `? :` is defined as follows. The expression `z ? x : y` has value  $x$  when  $z$  is *true* and has value  $y$  when  $z$  is *false*. As an example, the following ternary conditional statement and *if-else* statement are equivalent.

```
i = a > b ? a : b;           // ternary conditional statement

if (a > b) i = a;           // if-else statement
else i = b;
```

### *goto* Statements

It is sometimes necessary to make an explicit jump from the point of current execution to another point that has been labeled with an identifier using the *goto* statement:

```
if (condition) goto label;
label: statement
```

The labeled statement (a colon is needed after the label) can occur before or after the *goto* jump. A *goto* statement is useful when a jump is needed out of deeply nested blocks. For example,

```
repeat: i = 9;               // statement labeled by "repeat"
{
    // ...
    {
        // ...
        if (i == 0) goto repeat;
        if (i == 8) goto done;
    }
}
done: i = 1;                 // colon needed after label "done"
```

However, a succession of *goto* statements can lead to loss of control in a program and should be avoided. Even a single *goto* statement is rarely necessary.

### *switch* Statements

A *switch* statement is often used for multiple choice situations. It tests the value of its condition, supplied in parentheses after the keyword *switch*. Then this value will be used to switch control to a *case* label matching the value. If the value does not match any *case* label, control will go to the *default* case. The *break* statement is often used to exit a *switch* statement. For example,

```

int i, j, k;
// do something to compute the value of i

switch(i) {
    // i is tested against the cases
case 0:
    // control comes here if i == 0
    j = 6;
    k = 36;
    break;
    // branch out of switch statement
case 5:
    // control comes here if i == 5
    j = 7;
    k = 7;
    break;
    // branch out of switch statement
default:
    // use default if i != 0 && i !=5
    j = 8;
    // default statement
}

```

In this *switch* statement, the value of *i* is tested. When *i* is 0, the statement corresponding to *case 0* is executed. When *i* is 5, the statement corresponding to *case 5* is executed. When *i* is neither 0 nor 5, the default case is executed. Notice the appearance of the *break* statement after each *case*. Without it, every following statement in the *switch* statement would be executed, until a *break* statement is encountered. There are occasions when this is desired. A *default* choice is optional but is sometimes useful to catch errors. For example,

```

char x;
double y;
// do something to compute the value of x

switch(x) {
case 'A':
    // char enclosed in single quotes
case 'a':
    // when x is A or a, let y = 3.14
    y = 3.14;
    break;
    // branch out of switch statement
case 'B':
case 'b':
    // when x is B or b, let y = 2.17
    y = 2.17;
    break;
    // branch out of switch statement
default:
    // if x not A, a, B, b, report error
    cout << "Impossible value of x happened\n";
}

```

In general, the *switch* expression may be of any type for which integral conversion is provided and the *case* expression must be of the same type. A *switch* statement can always be replaced by an *if-else* statement, but a *switch* statement can sometimes make the code more readable.



### 2.3.7 Iteration Statements

#### *for* Loops

The *for* loop is very versatile and convenient to use. Its general form is

*for* (*initialize*; *condition*; *expression*) *statement*

where *initialize* represents statements initializing the control variable or variables, *condition* provides loop control to determine if *statement* is to be executed, and *expression* indicates how the variables initialized in *initialize* are to be modified and is evaluated (each time) after *statement* is executed. The loop proceeds until *condition* evaluates to *false*. If the first evaluation of *condition* is *false*, neither *expression* nor *statement* is evaluated at all. For example,

```
for (int i = 3; i < 50; i *= 2) {
    cout << i << '\n';
}
```

It will print out integers 3, 6, 12, 24, 48. The initial value of *i* is 3. Since condition *i* < 50 is *true*, 3 is printed. Then *i* \*= 2 gives the second value of *i* to be 6. Since *i* < 50 is still *true*, 6 is printed. This process repeats until *i* \*= 2 gives *i* = 96, at which time the condition *i* < 50 is no longer *true*.

The following loop will not print out anything since the initial condition is *false*.

```
for (int i = 3; i > 5; i *= 2) {
    cout << i << '\n';
}
```

When *i* = 3, the condition *i* > 5 is false. Thus the loop exits without executing the statement inside the braces. The expression *i* \*= 2 is also skipped.

A *for* loop defines a new *scope*. That is, the declaration *int i* = 3 in the previous *for* loop is out of scope outside the loop:

```
for (int i = 3; i < 50; i *= 2) { // i visible only in loop
    cout << i << '\n';
}
i = 20;                          // error, i out of scope
```

To avoid accidental misuse of a variable, it is usually a good idea to introduce the variable into the smallest scope possible. In particular, it is usually best to delay the declaration of a local variable until one can give it an initial value. Both *if-else* statements and *for* statements provide such a mechanism to avoid accidental misuse of variables and to increase readability.

The *break* statement can be used to stop a loop, and the *continue* statement stops the current iteration and jumps to the next iteration. For example,

```
for (int i = 3; i < 50; i *= 2) {
    if (i == 6) continue;    // jump to next iteration when i==6
    cout << i << '\n';
    if (i == 24) break;      // break the loop when i is 24
}
```

It will print out integers 3, 12, 24. When  $i$  is 6, the value of  $i$  is not printed out since the *continue* statement causes control to go to the next iteration with  $i = 6 * 2 = 12$ . When  $i$  is 24, the loop is exited due to the *break* statement. The *break* statement is often used to break an otherwise infinite loop. For example,

```
int n = 0;
for ( ; ; ) {                // an infinite loop
    cout << ++n << '\n';
    if ( n == 100) break;    // break infinite loop when n==100
}
```

It will print out 1, 2, ..., 100. The loop breaks when  $n$  is 100. This is an otherwise infinite loop since it has no condition to check.

Comma expressions sometimes appear inside a *for* loop and the statement of a *for* loop can be an empty statement. For example,

```
double sum;
int i;
for (sum = 0, i = 0; i < 100; sum += i, i++) ;
cout << "sum = " << sum << '\n';    // sum = 4950
```

It initializes  $sum = 0$  and  $i = 0$ . The comma expression  $sum += i, i++$  is executed until  $i < 100$  is not true. This *for* loop can also be written as

```
for (sum = 0, i = 0; i < 100; sum += i++) ;
```

The iteration variable in a *for* loop does not have to be integral. For example,

```
for (double x = 0; x < 300.14; x += 2.17) { sum += x; }
```

A *for* loop can be nested just like any other compound statement. For example,

```
double a = 0;
for (int i = 0; i < 100; i++) {
    for (int j = 200; j <= 500; j++) a += i - j;
}
```

```
long double b = 5;
for (int i = 1; i < 5; i++)
    for (int j = 27; j >= -3; j--) b *= i + j*j;
```

They compute, respectively,

$$\sum_{i=0}^{99} \sum_{j=200}^{500} (i - j) \quad \text{and} \quad 5 \prod_{i=1}^4 \prod_{j=-3}^{27} (i + j^2),$$

where  $\sum$  stands for summation and  $\prod$  for product.

### *while* Loops

The *while* loop statement has the form

*while (expression) statement*

The *statement* is executed until *expression* evaluates to *false*. If the initial value of *expression* is *false*, the *statement* is never executed. For example,

```
int x = 0;
while (x <= 100) {
    cout << x << '\n';
    x++;
}
```

The statements inside the braces are executed until  $x \leq 100$  is no longer true. It will print out 0, 1, 2, ..., 100. This loop is equivalent to the following *for* loop.

```
int x;
for (x = 0; x <= 100; x++) cout << x << '\n';
```

The following *while* loop will not print out anything since the initial condition is *false*.

```
int x = 10;
while (x > 100) {
    cout << x++ << '\n';
}
```

### *do-while* Loops

The *do-while* statement resembles the *while* loop, but has the form

*do statement while (expression);*

The execution of *statement* continues until *expression* evaluates to *false*. It is always executed at least once. For example,

```
int x = 10;
do {
    cout << x++ << "\n";
} while (x > 100);
```

It will stop after the first iteration and thus print out only 10.

The *continue* statement can also be used in a *while* or *do-while* loop to cause the current iteration to stop and the next iteration to begin immediately, and the *break* statement to exit the loop. Note that *continue* can only be used in a loop while *break* can also be used in *switch* statements. For example, the code segment

```
int x = 10;
do {
    cout << x++ << '\n';      // print x, then increment x
    if (x == 20) break;        // exit loop when x is 20
} while (true);
```

should print out all integers from 10 up to 19 and the code segment

```
int x = 10;
do {
    if (x++ == 15) continue;  // test equality, then increase x
    cout << x << '\n';
    if (x == 20) break;
} while (true);
```

should print out all integers from 11 up to 20 except for 16. However, the code

```
int x = 10;
while (true) {
    if (x == 15) continue;
    cout << x++ << '\n';
    if (x == 20) break;
}
```

first prints out integers 10, 11, 12, 13, 14 and then goes into an infinite loop (since  $x$  will stay equal to 15 forever).

## 2.4 Fibonacci Numbers

In this section, an example on Fibonacci numbers is given to illustrate the use of some C++ language features.

The sequence of Fibonacci numbers is defined recursively by

$$f_0 = 0, \quad f_1 = 1, \quad f_{n+1} = f_n + f_{n-1} \quad \text{for } n = 1, 2, 3, \dots$$

For instance,  $f_2 = f_1 + f_0 = 1 + 0 = 1$ , and  $f_3 = f_2 + f_1 = 1 + 1 = 2$ . Fibonacci numbers have many interesting applications and properties. One of the properties is that the sequence of Fibonacci quotients

$$q_n = f_n / f_{n-1}, \quad n = 2, 3, 4, \dots,$$

converges to the golden mean, which is  $(1 + \sqrt{5})/2 \approx 1.618$ .

The following program prints out the first 40 (starting with  $n = 2$ ) Fibonacci numbers and their quotients, using *long int* for Fibonacci numbers and *long double* for Fibonacci quotients.

Making use of the *for* loop, the program can be written as

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

main() {
    long fp = 1;           // previous Fibonacci number
    long fc = 1;           // current Fibonacci number

    for (int n = 2; n <= 40; n++) {           // main loop
        cout << n << "    ";           // output value of n
        cout << fc << "    ";           // output Fibonacci number
        cout << (long double)fc / fp << '\n'; // Fib quotient

        long tmp = fc;           // temporary storage
        fc += fp;                 // update Fib number
        fp = tmp;                 // store previous Fib number
    }
}
```

Inside the *for* loop, the variable *fc* is used to store the updated Fibonacci number and *fp* the previous one. Adding *fp* to *fc* gives the next Fibonacci number that is also stored in *fc*. A temporary storage *tmp* is used to store *fc* before the updating and assign the stored value to *fp* after updating, getting ready for the next iteration. Notice that the integer *fc* is converted into *long double* before the division *fc*/*fp* is performed, since otherwise the result would be just an integer.

This program works fine except the output may look messy. To have a formatted output, it may be modified to control the space in terms of the number of characters each output value occupies:

```
main() {
    long fp = 1;           // previous Fibonacci number
    long fc = 1;           // current Fibonacci number

    cout.width(2);           // output in a space of 2 chars
    cout << "n";
```

```

cout.width(27);           // output in a space of 27 chars
cout << "Fibonacci number";
cout.width(30);
cout << "Fibonacci quotient" << "\n\n";
cout.precision(20);       // precision for Fib quotients

for (int n = 2; n <= 40; n++) {
    cout.width(2);
    cout << n;
    cout.width(27);
    cout << fc;           // output Fibonacci number
    cout.width(30);
    cout << (long double)fc/ fp << '\n';    // Fib quotient

    long tmp = fc;        // temporary storage
    fc += fp;             // update Fib number
    fp = tmp;             // store previous Fib number
}
}

```

Some of the formatted output of the program is:

n	Fibonacci number	Fibonacci quotient
2	1	1
3	2	2
4	3	1.5
5	5	1.66666666666666666666
6	8	1.6
7	13	1.625
8	21	1.6153846153846153846
9	34	1.6190476190476190476
10	55	1.6176470588235294118
20	6765	1.6180339631667065295
30	832040	1.6180339887482036213
39	63245986	1.6180339887498951409
40	102334155	1.6180339887498947364

The results show that the Fibonacci quotients converge very quickly to the golden mean and the Fibonacci numbers grow extremely fast. Overflow will occur well before  $n$  reaches 100; see Exercise 2.5.14. A technique is introduced in Exercise 3.14.21 to store a large integer into an array of

integers digit by digit, which enables one to compute Fibonacci numbers for  $n$  up to thousands or larger.

## 2.5 Exercises

- 2.5.1. Add printing statements to the program at the end of §2.1.1 so that the value of each variable in different scopes is output to the screen right after it is defined. Check to see if the output values are the ones that you have expected.
- 2.5.2. The function `rand()`, declared in the standard header `<stdlib.h>`, generates a pseudorandom number between 0 and `RAND_MAX`. The macro `RAND_MAX` is also defined in `<stdlib.h>`. Write and run a program that outputs the value of `RAND_MAX` on your system and generates 100 pseudorandom numbers between 5 and 50, using a *for* loop.
- 2.5.3. Rewrite the program in Exercise 2.5.2 using a *while* loop. Compare the 100 numbers generated by this program and the program in Exercise 2.5.2.
- 2.5.4. Use a nested *for* loop to write a program to compute the sum:

$$\sum_{i=0}^{100} \sum_{j=5}^{300} \cos(i^2 + \sqrt{j}).$$

Check §3.11 for the cosine and square root functions in the math library `<math.h>`.

- 2.5.5. Write a program that computes the number of (decimal) digits of a given integer  $n$ . For example, when  $n = 145$ , the number of digits of  $n$  is 3.
- 2.5.6. Write a program that contains the statements

```
j = i-- + ++i;          // unpredictable result
k = j + j++;            // unpredictable result
k *= k++;               // unpredictable result
```

and outputs the values of  $j$  and  $k$  with  $i$  initialized to 5. Try to run the program on different machines if possible. The result of such statements is machine-dependent and can cause dangerous side effects. Experienced programmers can recognize such statements and do not use them.

- 2.5.7. The two statements

```
i = 5;
j = (i = 7) + (k = i + 3);
```

are legal, but give different results on different machines. The value of  $j$  is either 17 or 15 depending on whether the left subexpression ( $i = 7$ ) or the right subexpression ( $k = i + 3$ ) is evaluated first. The value of  $k$  is either 10 or 8. Test what values  $j$  and  $k$  have under your compiler. Again, such legal but machine-dependent statements should be avoided.

2.5.8. Can division by zero ever occur in the following code segment?

```
int x = rand(), y = rand();      // x, y are random int
if ((x != 5) && (y/(x-5) < 30))
    z = 2.17;                    // assume z declared
```

Then how about this (just switch the order of the subexpressions):

```
int x = rand(), y = rand();      // x, y are random int
if ((y/(x-5) < 30) && (x != 5))
    z = 2.17;                    // assume z declared
```

Note that the order of evaluation in logical expressions can make a difference.

2.5.9. Are the following two statements equivalent?

```
for (sum = 0, i = 0; i < 100; sum += i, i++) ;

for (sum = 0, i = 0; i < 100; i++, sum += i) ;
```

Note that it is guaranteed that evaluation is done from left to right in a comma expression. There are only three such expressions. What are they?

2.5.10. Run the following program:

```
main() {
    double sum = 0;
    for (double x = 0.0; x != 5.5; x += 0.1) sum += x;
}
```

It is intended to find the sum of  $0.1 + 0.2 + \cdots + 5.4$ . Does it go into an infinite loop on your computer? It does on most computers since  $x$  will never be equal to 5.5 due to computers' finite precision. In fact, numbers such as  $1/3$  and 0.1 can not be represented exactly on computers. To fix this *for* loop, replace it by



```
for (double x = 0.0; x <= 5.5; x += 0.1) sum += x;
```

Test this loop on your computer. Equality expressions involving floating point arithmetic must be used with great care. Replacing them by relational expressions often leads to more robust code.

- 2.5.11. Write a program to test the difference between the statement

```
while (i++ < n) { cout << i << '\n'; }
```

and the statement

```
while (++i < n) { cout << i << '\n'; }
```

where  $i$  and  $n$  are given integers, say  $i = 0$  and  $n = 10$ . Is there any difference between the following two statements?

```
for (char i = 'a'; i <= 'z'; i++) cout << i << '\n';
```

and

```
for (int i = 'a'; i <= 'z'; ++i) cout << i << '\n';
```

- 2.5.12. Here is an easy way to know the largest *unsigned int* on a computer with two's complement:

```
unsigned int largeui = - 1;
```

Test to see if *largeui* stores the largest *unsigned int* on your computer. Explain the reason. Does your compiler warn you about assigning a negative *int* to an *unsigned int*? If it does, how about:

```
int i = - 1;
unsigned int largeui = i;
```

- 2.5.13. Rewrite the first Fibonacci number program in §2.4 using a *while* loop and the second using a *do-while* loop so that they produce exactly the same output as the original programs.
- 2.5.14. Can you modify the program in §2.4 so that it will try to print out the first 100 Fibonacci numbers? Your output will probably have negative numbers for large  $n$ . This is caused by *integer overflow* since Fibonacci numbers grow very quickly. Computation usually continues but produces incorrect results when *overflow* occurs. It is the programmer's responsibility to make sure that all variables (suspected to take on large or small values) stay within range and stop the program or take other actions when *overflow* happens. Rewrite the program so that it will print out a warning message before *overflow* occurs.

- 2.5.15. Write a program that outputs the bit representation of an integer. On a computer with  $\text{sizeof}(\text{int}) = 4$  (i.e., 32 bits are used to represent an integer) the program may be written as

```
main() {
    int n;
    int mask = 1 << 31;    // mask = 100000000 ... 000000000
    for (; ; ) {          // enter an infinite loop
        cout << "\nEnter an integer: \n";    // input an int
        cin >> n;          // input is assigned to n

        cout << "bit representation of " << n << " is: ";
        for (int i = 1; i <= 32; i++) { // loop over 32 bits
            char cc = (n & mask) ? '1': '0';
                                // take bit i from left to right
            cout << cc;    // print bit i from left to right
            n <<= 1;        // n = n << 1; left shift n by 1 bit
            if (i%8 == 0 && i != 32) cout << ' ';
                                // print one space after each byte
        }
    }
}
```

Note that  $n \& \text{mask}$  gives the leftmost bit of  $n$  followed by 31 '0's in bit representation. When this bit is 1, character '1' is assigned to  $cc$  and otherwise  $cc = '0'$ . The statement  $n \<\<= 1$  shifts the bit pattern of  $n$  to the left by 1 bit. When  $i = 1$ , the leftmost bit of  $n$  is printed out and when  $i = 2$  the second leftmost bit is printed out, and so on. Run the program on your computer and print out the bit representations of 55555 and  $-55555$ . Modify the program to handle integers with  $\text{sizeof}(\text{int}) = m$ , where  $m$  may be different on different machines.

# 3

## Derived Types

From basic types, other types can be derived. This chapter discusses derived types such as enumeration types for representing a specific set of values, pointers for manipulating addresses or locations of variables, arrays for storing a number of elements having the same type, reference types for creating aliases, structures for storing elements of different types, and unions for storing elements of different types when only one of them is present at a time. Functions—a special type—that may take some values as input and produce some other values as output are also discussed. Another derived type called *class* is discussed in Chapter 5. All C++ operators and their precedences, and all mathematical functions in the standard library *<math.h>* are summarized towards the end of the chapter. The chapter ends with one section on polynomial evaluations and another on numeric integration techniques.

### 3.1 Constants and Macros

A value that can not be changed throughout a program is called a *constant*. It is defined using the keyword *const*. It is illegal to redefine a constant and for this reason, a constant should be initialized when it is declared. For example,

```

const char tab = '\t';           // tab character
const double pi = 3.1415926535897932385; // pi
const double e = 2.7182818284590452354; // natural log base

```

```

pi = 3.1;           // illegal, a const can not be changed
const int size;     // illegal, a const must be initialized

const int dimension = 3;
const int numelements = 100*dimension; // const expression

```

Note that *const* changes a type in that it restricts the way in which an object can be used. Constant expressions are allowed in the definition of a constant such as *numelements* above, since the compiler can evaluate constant expressions.

Constants are typed and thus preferable to traditional C macros, which are not type-checked and are error-prone, such as

```
#define PI 3.1415926535897932385
```

A macro is defined by using *define* preceded by *#*, which must stand in the first column. A preprocessor will substitute every occurrence of *PI* by 3.1415926535897932385 in the program when it is compiled. An example of a macro with an argument is:

```
#define SQ(x) ((x) * (x))
```

Then every occurrence of *SQ(x)* will be substituted by *((x) \* (x))* in the program. For example, *SQ(a + b)* will be substituted by *((a + b) \* (a + b))*. Notice the necessity of the parentheses in the definition of a macro. Otherwise *SQ(a + b)* would be just *(a + b \* a + b)* if the two inner pairs of parentheses were omitted in the macro definition. By convention, capital letters are usually used for macro names. Although macros are still legal in C++, they are best avoided.

The *const* specifier can be used together with the keyword *volatile*. A *volatile* specifier is a hint to the compiler that an object may be modified in ways not detectable by the compiler and aggressive optimizations must be avoided. For example, a real-time clock may be declared as:

```
extern const volatile int clock;
```

This says that *clock* may be changed by an agent external to the program since it is *volatile* and may not be modified inside the program since it is *const*. Thus two successive reads of the variable *clock* in the program may give two different values. An *extern const* may not have to be initialized.

## 3.2 Enumerations

The enumeration type *enum* is for holding a set of integer values specified by the user. For example,

```
enum {blue,yellow,pink =20,black,red =pink +5, green =20};
```

is equivalent to

```
const int blue = 0, yellow = 1, pink = 20, black = 21,
        red = 25, green = 20;
```

By default, the first member (enumerator) in an *enum* takes value 0 and each succeeding enumerator has the next integer value, unless other integer values are explicitly set. The constant *pink* would take value 2 if it were not explicitly defined to be 20 in the definition. The member *black* has value 21 since the preceding member *pink* has value 20. Note that the members may not have to take on different values.

An enumeration can be named and thus create a distinct type. By default enumerations can be implicitly converted to integers, but there is no implicit conversion from integers to enumerations to avoid unpredictable results. For example,

```
enum bctype {Dirichlet, Neumann, Robin}; // a named enum
bctype x, y; // declare x, y to be of type bctype
x = Dirichlet; // assigning a value to variable x
y = 0; // illegal, an int can not be assigned
y = 40; // to a variable of type bctype

x = bctype(2); // OK, explicit conversion, x = Robin
int i = Neumann; // OK, enum is restricted int, i = 1

if (x != y) i = Robin; // they can be compared
```

The first statement above defines a new type called *bctype*. A variable of this type may take on values of *Dirichlet*, *Neumann*, and *Robin*.

An *enum* is a user-defined type and often used in conjunction with the *switch* statement:

```
bctype x; // declare x to be of type bctype

// ... compute or get the value of x

switch(x) { // do differently based on value of x
case Dirichlet:
    // ... deal with Dirichlet boundary condition
    break;
case Neumann: case Robin:
    // ... deal with Neumann and Robin conditions at same time
    break;
Default:
    cout << "undefined boundary condition\n";
}
```

Although a programmer rarely needs to know the range of an enumeration type, the range is normally larger than the set of values of the enumerators. The range is defined to be the set of all integers in the closed interval  $[0, 2^n - 1]$  if the smallest enumerator is nonnegative and in the closed interval  $[-2^n, 2^n - 1]$  if the smallest enumerator is negative, where  $n$  is the nearest larger binary exponent of the largest (in magnitude) enumerator. This is the smallest bit-field that can hold all the enumerator values. For example,

```
enum flag { red = 1, blue = 0 };      // range: 0, 1
enum onenine { one = 1, nine = 9 };  // range: 0, 1, ..., 15
enum myminmax{ min = -90, max = 1000};
                                     // range: - 2048, ..., 2047

onenine m0 = onenine(5);              // OK, 5 is within range
myminmax m1 = myminmax(- 1000);       // OK, -1000 within range
myminmax m2 = myminmax(5000);         // error, out of range
```

Since  $9 = 2^3 + 1$ , the nearest larger binary exponent is  $n = 4$ . Thus the range of *onenine* is the set of all integers from 0 up to  $2^4 - 1 = 15$ .

Constants and enumerations are designed for a better type-checking mechanism to avoid possible type errors and accidental modification of constant values.

### 3.3 Arrays

For a type  $T$ ,  $T[n]$  is the type “one-dimensional array of  $n$  elements of type  $T$ ,” where  $n$  is a positive integer. The elements are indexed from 0 to  $n - 1$  and are stored contiguously one after another in memory. For example,

```
float vec[3];      // array of 3 floats: vec[0],vec[1],vec[2]
int stg[30];       // array of 30 ints: stg[0],..., stg[29]

vec[0] = 1.0;      // accessing element 0 of vec
vec[1] = 2.0;      // accessing element 1 of vec

for (int i = 0; i < 30; i++) stg[i] = i*i + 7;
int j = stg[29];   // accessing the last element of stg
```

The first two statements declare *vec* and *stg* to be one-dimensional arrays with 3 and 30 elements of type *float* and *int*, respectively. A *for* loop is often used to access all elements of a 1D array. A one-dimensional array can be used to store elements of a (mathematical) vector.

Two-dimensional arrays having  $n$  rows and  $m$  columns (looking like a matrix) can be declared as  $T[n][m]$ , for elements of type  $T$ . The row index changes from 0 to  $n - 1$  and the column index from 0 to  $m - 1$ . For example,

```
double mt[2][5]          // 2D array of 2 rows and 5 columns
mt[0][0] = 5;           // access entry at row 0, column 0
mt[1][4] = 5;           // access entry at row 1, column 4
double a = mt[0][0];    // access entry [0][0] of mt

for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 5; j++) mt[i][j] = i + j;
}
```

The first statement above declares *mt* to be a two-dimensional array with 2 rows and 5 columns, whose element at row *i* and column *j* can be accessed by using *mt[i][j]*. A nested *for* loop is often used to access all elements of a 2D array. A two-dimensional array can be thought of as a one-dimensional array of one-dimensional arrays. For example, *mt* can be viewed as having two “elements”: *mt[0]* and *mt[1]*, where *mt[0]* is a one-dimensional array of five elements representing the initial row (row 0) of *mt*, and *mt[1]* is also a one-dimensional array of five elements but representing the second row (row 1). The symmetric notation *mt[i][j]* seems to be superior to FORTRAN’s notation *mt(i, j)* since *mt[i]* can be used as a one-dimensional array and element *j* of *mt[i]* is *mt[i][j]*. A two-dimensional array is a natural data structure for storing elements of a matrix.

Three and more dimensional arrays can be declared similarly. Let  $s_1, s_2, \dots, s_k$  be positive integers; then the declaration of a *k*-dimensional array *a[s<sub>1</sub>][s<sub>2</sub>] ... [s<sub>k</sub>]* will allocate spaces for  $s_1 \times s_2 \times \dots \times s_k$  elements, which are stored contiguously in memory row by row (unlike in FORTRAN, where they are stored column by column).

Arrays can be initialized at the time of declaration. For example,

```
int v[] = {1, 2, 4, 5};      // initialization of 1D array
int a[3] = {2, 4, 5};       // initialization of 1D array
int u[ ][3] = { {1, 2, 3}, {4, 5, 8} };
                           // initialization of 2D array
char filename[30] = "outpt"; // a character array
```

In the declaration of *v* above, the compiler will count the number of elements in the initialization and allocate the correct amount of space for *v* so that the number of elements can be omitted. In an initialization of a multi-dimensional array such as *u[ ][3]*, only the first dimension (the number of rows for *u*) can be omitted. However, assignments can not be done this way:

```
int vv[4];                  // a declaration of a 1D array
vv = {1, 2, 4, 5};          // error, use: vv[0] =1, vv[1] =2, ...
```

Since multidimensional arrays are stored row by row, they can be initialized like one-dimensional arrays. For example, the following two initializations are equivalent.

```
int ia[2][3] = {1, 2, 0, 4, 0, 0};
int ia[2][3] = { {1, 2, 0}, {4, 0, 0} };
```

Entries that are not explicitly initialized are set to 0. Thus they are also equivalent to:

```
int ia[2][3] = { {1, 2}, {4} };
```

For this reason, the initialization

```
int ib[2][3] = {5};
```

initializes the first entry  $ib[0][0] = 5$  and all other entries to 0.

Here is an example of an initialization of a three-dimensional array:

```
double ax[4][2][3] = {
    { {11, 22, 0}, {55, 0, 0} }, // ax[0]
    { {-1, -2, 0}, {-5, -6, 0} }, // ax[1]
    { {23, -7, 8}, {0, 13, 8} }, // ax[2]
    { {-3, 19, 0}, {9, -5, 3} } // ax[3]
};
```

In particular,  $ax[0][0][0] = 11$ ,  $ax[2][0][2] = 8$ ,  $ax[3][1][1] = -5$ , and  $ax[0]$  is a two-dimensional array with entries  $\{\{11, 22, 0\}, \{55, 0, 0\}\}$ . Its entries may be accessed by a triple loop:

```
for (int i = 0; i < 4; i++)
    for (int j = 0; j < 2; j++)
        for (int k = 0; k < 3; k++) cout << ax[i][j][k] << '\n';
```

The dimensions of an array must be known at the time it is declared so that the compiler can allocate space for it. Otherwise it is illegal. For example,

```
const int m = 2000;
const int k = 2*m + 100; // compiler evaluates k = 4100
double b[k];             // OK, k is known to compiler

// ... compute the value of n at run-time
double a[n];             // error, compiler does not know value of n
```

The number of elements in a sequence often can not be known at compile-time, but rather is calculated at run-time. If we declare a large array, it wastes space. If we declare a small one, it may not be big enough to hold all the elements. For this reason arrays should not be used unless their dimensions are exactly or roughly known at compile-time. Instead, pointers are often used and space can be allocated at run-time; see §3.6. The standard libraries `<valarray>` and `<vector>` can also be conveniently used to handle arrays whose sizes may not be known at compile-time; see §7.5 and §10.1.1.



## 3.4 Structures

Unlike an array that takes values of the same type for all elements, a *struct* can contain values of different types. It can be declared as

```
struct point2d {           // a structure of 2D points
    char nm;               // name of the point
    float x;               // x-coordinate of point
    float y;               // y-coordinate of point
};
```

This defines a new type called *point2d*. Note the semicolon after the right brace. This is one of the very few places where a semicolon is needed following a right brace. The structure *point2d* has three fields or members, one *char* and two *floats*. Its size in memory is *sizeof(point2d)* bytes, which may not be equal to *sizeof(char) + 2 \* sizeof(float)*. Its members are accessed by the . (dot) operator. For example,

```
point2d pt;                // declare pt of type point2d
pt.nm = 'f';               // assign char 'f' to its field nm
pt.x = 3.14;               // assign 3.14 to its field x
pt.y = - 3.14;             // assign -3.14 to its field y

double a = pt.x;           // accessing member x of pt
char c = pt.nm;            // accessing member nm of pt
```

A variable of a *struct* represents a single object and can be initialized by and assigned to another variable (consequently, all members are copied). For example,

```
point2d pt2 = pt;          // initialize pt2 by pt, memberwise copy
pt = pt2;                  // assign pt2 to pt, memberwise copy
```

It can also be initialized in a way similar to arrays:

```
point2d pt3 = {'F', 2.17, - 3.14}; // OK, initialization

point2d pt4;
pt4 = {'A', 2.7, - 3.4};           // illegal, assignment
```

An example of *struct* with an unnamed *enum* as a member can be found in §7.7.4.

## 3.5 Unions and Bit Fields

Unions are to save space, but follow the same syntax as structures. Only one member of a *union* can exist at a time and all members take up only as much space as its largest member. It can be declared as

```

union val {           // i, d, c can not be used at same time
    int i;
    double d;         // d is largest member in storage
    char c;
};

```

The union *val* has three members: *i*, *d*, and *c*, only one of which can exist at a time. Thus *sizeof(double)* bytes of memory are enough for storing an object of *val*. The exact size of *val* is *sizeof(val)*. Members of a union are also accessed by the . (dot) operator. It can be used as the following.

```

int n;
cin >> n;              // n is taken at run-time

val x;                  // x is a variable of type val
if (n == 1) x.i = 5;
else if (n == 2) x.d = 3.14;
else x.c = 'A';

double v = sin(x.d);    // error, x.d may not exist all times

```

When *x* is declared, *sizeof(double)* bytes of memory are created for it. This piece of memory can be used to store an *int* (member *x.i*), a *double* (member *x.d*), or a *char* (member *x.c*). The last statement above is an error (a compiler can not catch such an error and a programmer should be responsible for this) since *x.d* may not be defined at all times. To see this, look at the program:

```

main() {

    union val {
        int i;
        double d;
        char c;
    };

    val x;
    x.i = 5;              // only x.i is defined now.
                        // Other members should not exist
    cout << "x.i = " << x.i << ", x.d = "
         << x.d << ", x.c = " << x.c << '\n';

    x.d = 6.28;           // only x.d is defined now.
    cout << "x.i = " << x.i << ", x.d = "
         << x.d << ", x.c = " << x.c << '\n';

    x.c = 'G';            // only x.c is defined now.
}

```

```

    cout << "x.i = " << x.i << ", x.d = "
          << x.d << ", x.c = " << x.c << '\n';
}

```

The output of this program on my computer is:

```

x.i = 5, x.d = 3.10143, x.c =
x.i = 1374389535, x.d = 6.28, x.c =
x.i = 1374389575, x.d = 6.28, x.c = G

```

Note that only the member that is defined can be correctly printed out. Values of other members should not be used. When *x.i* is defined to be 5, its value can be printed out (and used) correctly. However, this same piece of memory is interpreted to be a *double* and a *char* when *x.d* and *x.c* are printed out, respectively. The outputs for *x.d* and *x.c* are of course erroneous in the first printing statement above.

Suppose that *triangle* and *rectangle* are two structures and a *figure* can be either a triangle or a rectangle but not both; then a structure for *figure* can be declared as

```

struct figure2d {
    char name;
    bool type;           // 1 for triangle, 0 for rectangle
    union {              // an unnamed union
        triangle tria;
        rectangle rect;
    };
};

```

If *fig* is a variable of type *figure2d*, its members can be accessed as *fig.name*, *fig.type*, *fig.tria*, or *fig.rect*. Since a figure can not be a rectangle and a triangle at the same time, using a *union* can save memory space by not storing *triangle* and *rectangle* at the same time. The member *fig.type* is used to indicate if a triangle or rectangle is being stored in an object *fig* (e.g., *fig.rect* is defined when *fig.type* is 0). Such a member is often called a *type tag*.

Note that a union without a name is used inside *figure2d*. If a named union is used, for example,

```

struct figure2d {
    char name;
    bool type;           // 1 for triangle, 0 for rectangle
    union torr {         // a named union
        triangle tria;
        rectangle rect;
    };
};

```

then the members *tria* and *rect* will have to be accessed as *fig.torr.tria* and *fig.torr.rect*, for a *figure2d* object *fig*.

The size of a C++ object is a multiple of the size of a *char*, which is one byte. That is, a *char* is the smallest object that can be represented on a machine. There are occasions when it is necessary to put a few small objects into one byte to squeeze more space. A member of a *struct* or *union* can be declared to occupy a specified number of bits. Such a member is called a *bit field* and the number of bits it occupies is called its *bit width*. A bit field must be of an integral type (typically *unsigned int*), and it is not possible to take its address. The bit width is at most the number of bits in a machine word and is specified by a nonnegative constant integral expression following a colon. Unnamed bit fields are also allowed for padding and alignment. An example is

```
struct card {
    unsigned int pips : 4;    // pips occupies 4 bits
    unsigned int suit : 2;    // suit occupies 2 bits
    unsigned int kq  : 2;    // kq occupies 2 bits
    // ...                  // other fields can be added
};
```

A variable of *card* has a 4-bit field *pins* that can be used to store small nonnegative integers 0 to 15, a 2-bit field *suit* for values 0 to 3, and another 2-bit field *kq*. Thus the values of *pins*, *suit*, and *kq* can be compactly stored in 8 bits. It can be used as

```
card c;                // c is of type card
c.pips = 13;
c.suit = 3;
c.kq   = 0;
```

The layout of the bits in left-to-right or right-to-left order is machine-dependent. A compiler may not assign bit fields across word boundaries. For example, *sizeof(card)* = 4 on some machines with 4-byte words. (It still saves memory in this case since storing three *unsigned int* separately would require 12 bytes.) Programs manipulating bits may not be portable. Unless saving memory is really necessary for an application, bit fields (also unions) should be avoided.

## 3.6 Pointers

For a type *T*, *T\** is the pointer to *T*. A variable of type *T\** can hold the address or location in memory of an object of type *T*. For this reason, the number of bytes of any pointer is *sizeof(int\*)* = *sizeof(double\*)*, and so on. For example, the declaration

```
int* p;           // p is a pointer to int
```

declares the variable  $p$  to be a pointer to  $int$ . It can be used to store the address in memory of integer variables.

If  $v$  is an object,  $\&v$  gives the address of  $v$  (the address-of operator  $\&$  is introduced in §2.3.1). If  $p$  is a pointer variable,  $*p$  gives the value of the object pointed to by  $p$ . We also informally say that  $*p$  is the value pointed to by  $p$ . The operator  $*$  is called the *dereferencing* or *indirection* operator. The following example illustrates how the address-of operator  $\&$  and the *dereferencing* operator  $*$  can be used:

```
int i = 5;           // i is int, value of object i is 5
int* pi = &i;        // pi is a pointer to int
                    // and assign address of i to pi

int j = *pi;         // value of object pointed to by pi
                    // is assigned to j, so j=5

double* d = &j;      // illegal
```

The second statement above declares  $pi$  to be a variable of type: pointer to  $int$ , and initializes  $pi$  with the address of object  $i$ . Another way of saying that pointer  $pi$  holds the address of object  $i$  is to say that pointer  $pi$  points to object  $i$ . The third statement assigns  $*pi$ , the value of the object pointed to by  $pi$ , to  $j$ . The fourth statement is illegal since the address of a variable of one type can not be assigned to a pointer to a different type.

For a pointer variable  $p$ , the value  $*p$  of the object that it points to can change; so can the pointer  $p$  itself. For example,

```
double d1 = 2.7, d2 = 3.1;
double* p = &d1;      // p points to d1, now *p = 2.7
double a = *p;        // a = 2.7

p = &d2;              // p now points to d2, *p = 3.1
double b = *p;        // b = 3.1

*p = 5.5;             // value p points to is now 5.5
double c = *p;        // c = 5.5
double d = d2;        // d = 5.5, since *p = 5.5
```

Since  $p$  is assigned to hold the address of  $d2$  in the statement  $p = \&d2$ , then  $*p$  can also be used to change the value of object  $d2$  as in the statement  $*p = 5.5$ . When  $p$  points to  $d2$ ,  $*p$  refers to the value of object  $d2$  and assignment  $*p = 5.5$  causes  $d2$  to equal 5.5.

A sequence of objects can be created by the operator *new* and the address of the initial object can be assigned to a pointer. Then this sequence can be used as an array of elements. For example,

```

int n = 100;           // n can also be computed at run-time
double* a;             // declare a to be a pointer to double
a = new double [n];    // allocate space for n double objects
                      // a points to the initial object

```

The last two statements can also be combined into a more efficient and compact declaration with an initialization:

```
double* a = new double [n]; // allocate space of n objects
```

In allocating space for new objects, the keyword *new* is followed by a type name, which is followed by a positive integer in brackets representing the number of objects to be created. The positive integer together with the brackets can be omitted when it is 1. This statement obtains a piece of memory from the system adequate to store  $n$  objects of type *double* and assigns the address of the first object to the pointer *a*. These objects can be accessed using the array subscripting operator `[]`, with index starting from 0 ending at  $n - 1$ . For example,

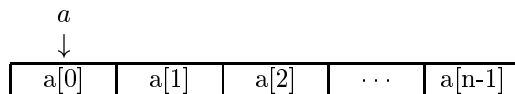
```

for (int i = 0; i < n; i++) // a[0] refers to initial object
    a[i] = i*i + 8;         // a is used just like an array

double sum = 0;             // find sum of all objects a[i]
for (int i = 0; i < n; i++) sum += a[i];

```

The pictorial representation of the pointer *a* and the space allocated for it can be drawn as



You can print out the values of `*a` and `a[0]` to check that they are indeed the same. This agrees with the fact that *a* points to the initial object `a[0]`. After their use, these objects can be destroyed by using the operator *delete* :

```
delete[] a;           // free space pointed to by a
```

The system will automatically find the number of objects pointed to by *a* (actually *a* only points to the initial object) and free them. Then the space previously occupied by these objects can be reused by the system to create other objects. Since the operator *new* creates objects at run-time, this is called *dynamic memory allocation*. The number of objects to be created by *new* can be either known at compile-time or computed at run-time, which is preferred over the built-in arrays (§3.3) in many situations. In contrast, creation of objects at compile-time is called *static memory allocation*. Thus there are two advantages of dynamic memory allocation: objects no longer

in use can be deleted from memory to make room to create other objects, and the number of objects to be created can be computed at run-time.

Automatic variables represent objects that exist only in their scopes. In contrast, an object created by operator *new* exists independently of the scope in which it is created. Such objects are said to be on the *dynamic memory* (the *heap* or the *free store*). They exist until being destroyed by operator *delete* or to the end of the program.

Space can also be allocated for a pointer to create just one object in a similar way:

```
char* pt;           // pt is a pointer to character
pt = new char;      // allocate space of 1 char for pt
pt[0] = 'c';        // place 'c' at location pointed to by pt
*pt = 'd';          // place 'd' at location pointed to by pt
char x = pt[0];     // assign pt[0] to object x, x = 'd'
char y = *pt;       // assign *pt to object y, y = 'd'
delete pt;          // free space pointed to by pt
```

Note that when more than one object is freed, *delete[]* should be used; otherwise use *delete* without brackets.

An object can also be initialized at the time of creation using *new* with the initialized value in parentheses. For example,

```
double* y = new double (3.14); // *y = 3.14
int i = 5;
int* j = new int (i); // *j = 5, but j does not point to i
```

Declarations of forms  $T * a$ ; and  $T * a$ ; are equivalent, as in

```
int* ip;           // these two declarations are equivalent
int *ip;
```

However, the following two declarations are equivalent,

```
int* i, j;        // i is a pointer to int but j is an int
int *i, j;
```

each of which declares *i* to be a pointer to *int* and *j* to be an *int*. This kind of inconsistency is inherited from C, so that many C functions can be called from C++. To avoid confusion, it is best to declare a pointer variable in a single declaration statement.

An array of pointers and a pointer to an array can also be defined:

```
int* ap[10] ;      // ap is an array of 10 pointers to int
int (*vp)[10];     // vp is a pointer to an array of 10 int
```

Notice that parentheses are needed for the second statement above, which declares *vp* to be a pointer to an array of 10 integers. The first statement declares *ap* to be an array of 10 pointers, each of which points to an *int*.

### 3.6.1 Pointer Arithmetic

Some arithmetic operations can be performed on pointers. If  $pt$  is a pointer, then  $*pt$  represents the object to which  $pt$  points. If  $pt$  is a pointer and space has been allocated to store more than one object, then  $pt$  points to object 0 (the initial object) and  $pt + k$  points to object  $k$  ( $k$ th object counted starting from 0). Notice  $pt + k$  is a special arithmetic operation that simply refers to object  $k$  in the sequence. The value of object  $k$  is  $*(pt + k)$  or  $pt[k]$ . For example,

```
char* cc;           // cc is a pointer to char
cc = new char [3];  // allocate 3 char for cc
                    // cc points to initial char

cc[0] = 'a';        // or *cc = 'a'; cc points to object 0
cc[1] = 'b';        // or *(cc + 1) = 'b';
                    // (cc + 1) points to object 1

cc[2] = 'c';        // or *(cc + 2) = 'c';
                    // (cc + 2) points to object 2

char* p = &cc[0];   // assign address of cc[0] to p
char dd = *p;       // dd = 'a', since *p = cc[0]
dd = *(cc + 2);     // (cc+2) points to 'c', now dd = 'c'

delete[] cc;        // free space allocated by new
```

The object pointed to by  $p$  is  $cc[0]$  and the value of  $cc[0]$  is 'a'. Thus the value of the object pointed to by  $p$ , that is,  $*p$ , is 'a', which is initialized to  $dd$ . Similarly,  $cc + 2$  points to object 2 in the sequence and thus  $*(cc + 2)$  is the value of object 2, which is also referred to by  $cc[2]$ . The term *element* instead of object is often used when referring to an individual in a sequence, such as an array.

### 3.6.2 Multiple Pointers

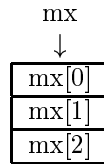
Two-dimensional arrays and matrices can be achieved through double pointers (a pointer to a pointer is called a *double pointer*). For example,

```
int** mx;           // double pointer: a pointer to a pointer
mx = new int* [n];  // new space to hold n pointers to int
                    // mx points to initial element mx[0]

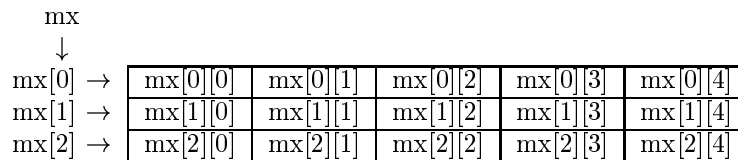
for (int i = 0; i < n; i++) mx[i] = new int [m];
                    // create m objects for each of the n pointers
                    // mx[i] points to initial element mx[i][0]
```



The first statement above declares *mx* to be a pointer to a pointer, called a *double pointer*. The second statement allocates *n* objects of type *int\** and assigns the address of the initial element to *mx*. It happens that these *n* objects are pointers to *int*. Now *mx* has value `&mx[0]`. This can be pictorially depicted in the case of *n* = 3 as



The third statement above (the *for* loop) allocates *m* objects of type *int* for each of the *n* pointers *mx[i]*, *i* = 0, 1, ..., *n* - 1. A graphical representation of the double pointer *mx* in the case of *n* = 3 and *m* = 5 can now be drawn as



The process above may be more easily explained using the keyword *typedef*. A declaration prefixed by *typedef* introduces a new name for a type rather than declaring a new variable. For example, the statement

```
typedef int* intptr;           // intptr is synonym to int*
```

introduces *intptr* as a synonym to pointer to *int*. A *typedef* does not introduce a new type so that *intptr* means the same as *int\**. Then the double pointer *mx* can be equivalently declared as

```
intptr* mx = new intptr [n];
for (int i = 0; i < n; i++) mx[i] = new int [m];
```

It says that *mx* is a pointer to *intptr*, and *n* objects: *mx[i]*, *i* = 0, 1, ..., *n* - 1, of type *intptr* are created by using the *new* operator. Then *m* objects of type *int* are created for each *mx[i]*.

Now *mx* can be used in the same way as a 2D array:

```
for (int i = 0; i < n; i++) // mx[i] is a pointer to int
    for (int j = 0; j < m; j++) // used just like a 2D array
        mx[i][j] = i*i + 9;    // access element mx[i][j]
```

After use, the space can be freed as

```

for (int i = 0; i < n; i++)    // free space after use
    delete[] mx[i];           // first free these n pointers
delete[] mx;                   // then free double pointer mx

```

In the above,  $n$  and  $m$  can be computed at run-time and space for  $mx$  can be allocated dynamically. This is in contrast to an array that can not be declared unless its dimension is known at compile-time. Note that the orders of allocating and deleting space for the double pointer  $mx$  are opposite. The same rule applies to other multidimensional pointers. The effect of the double pointer  $mx$  can be viewed as a matrix:  $mx$  is a pointer that points to the initial object  $mx[0]$  of an array of  $n$  pointers:  $mx[i]$ ,  $i = 0, 1, \dots, n-1$  (the rows of the matrix). Each of the  $n$  pointers subsequently points to the initial object  $mx[i][0]$  of an array of  $m$  integers:  $mx[i][j]$ ,  $j = 0, 1, \dots, m-1$  (each row has  $m$  elements). The element at row  $i$  and column  $j$  can be accessed using  $mx[i][j]$ , the same way as in a 2D array. It can also be alternatively accessed using the dereferencing operator as:  $*(mx[i] + j)$ ,  $((mx + i)[j])$ ,  $*(mx + i)[j]$ , or  $(&mx[0][0] + m * i + j)$ .

Using pointers an  $n$  by  $n$  lower triangular or symmetric matrix can be defined very conveniently (to save memory, zero or symmetric elements above the main diagonal are not stored):

```

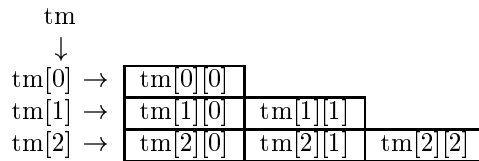
double** tm = new double* [n];
for (int i = 0; i < n; i++) tm[i] = new double [i+1];
                                // allocate (i+1) elements for row i

for (int i = 0; i < n; i++)    // access its elements
    for (int j = 0; j <= i; j++) tm[i][j] = 2.1/(i+j+1);

for (int i = 0; i < n; i++) delete[] tm[i];
delete[] tm;                   // after using it, delete space

```

In the above,  $tm$  is created to store an  $n$  by  $n$  lower triangular matrix. Since the lower triangular part of a matrix contains  $i+1$  elements in row  $i$  for  $i = 0, 1, \dots, n-1$ , only  $i+1$  doubles are allocated for  $tm[i]$ . A picture for illustrating this idea can be drawn as (when  $n = 3$ )



Upper triangular matrices can also be declared using the techniques discussed here and in §3.6.3; see Exercise 3.14.12. Note that arrays can only represent rectangular matrices. Using rectangular matrices to store triangular matrices or symmetric matrices would waste space.

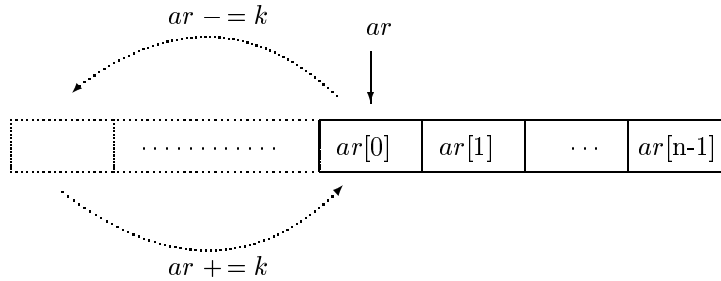


FIGURE 3.1. Pointer offsetting. Originally,  $ar$  points to initial object  $ar[0]$ . Pointer arithmetic  $ar -= k$  offsets  $ar$  to the left by  $k$  positions. Then,  $ar += k$  changes  $ar$  back to point to the initial object. That is, after offsetting,  $ar[k] = *(ar + k)$  is the initial object.

See Exercise 3.14.14 for triple and quadruple pointers, which can be used to dynamically allocate space for three- and four-dimensional arrays. Higher-dimensional arrays may be created similarly, but are less often used. Double, triple, and quadruple pointers, and the like, may be called *multiple pointers*.

### 3.6.3 Offsetting Pointers

It is sometimes very useful to use an array of  $n$  elements whose index changes from  $k$  to  $n - 1 + k$ , where  $k$  is a positive or negative integer. The case of  $k = 0$  corresponds to a regular array, where the index changes from 0 to  $n - 1$ . The idea is to offset the pointer,  $k$  elements away from its original place. For example, let us first assume that  $k$  is positive:

```
double* ar = new double [n]; // ar points to 1st element
ar -= k; // offset ar, k elements to the left
```

The first statement allocates space for a sequence of  $n$  elements and  $ar$  points to the initial element in the sequence. The second statement is simply a pointer arithmetic operation, which causes  $ar$  to be decremented by  $k$ . In other words, pointer  $ar$  is offset by  $k$  positions to the left. Now  $ar$  may not point to any valid object at all and  $ar[0]$  should never be used. However, if  $ar$  is incremented by  $k$ , it should come back and refer to the initial element in the sequence. That is, now  $*(ar + k) = ar[k]$  represents the initial element in the original sequence, and subscripting starting from  $k$  has been achieved. See Figure 3.1 for a pictorial description.

The  $n$  objects can now be referred to as

```
for (int i = k; i < n + k; i++) {
```

```

    ar[i] = i*i + 1;      // now ar[k] is initial element
                          // ar[n+k-1] is the last element
}

```

Note that the objects are not moved at all in this process. Only the pointer *ar* is offset (subtracted by *k*) and changed back (added by *k*) to point to the same object. When *k* is negative, the pointer is actually offset  $|k|$  elements to the right and the code above remains exactly the same. When the memory is reclaimed, the pointer needs to be changed back first:

```

ar += k;                // first change pointer ar back
delete[] ar;            // then delete it as usual

```

For example, to store the number of students in classes 1999 and 2000 in an array of 2 elements, the code can be written as

```

int* ns = new int [2];  // ns points to initial element
ns -= 1999;             // offset ns 1999 elements to left
ns[1999] = 36000;       // ns[1999] now is initial element
ns[2000] = 37000;       // 37000 students in class 2000
ns += 1999;            // change ns back before deleting
delete[] ns;           // delete the space

```

If an array of 108 elements is needed, but the index must change from  $-100$  to  $7$ , it can be done as

```

double* ade = new double [108];
ade += 100;      // offset ade 100 elements to right
for (int i = - 100; i <= 7; i++)
    ade[i] = i + 9.8; // access its elements
ade -= 100;      // change ade back before deleting
delete[] ade;    // delete the space

```

This technique is extremely useful and convenient when dealing with band matrices, where the nonzero elements of a matrix form a band along the main diagonal. See Chapter 11.

#### 3.6.4 Constant Pointers

A *constant pointer* is a pointer that can not be redefined to point to another object; that is, the pointer itself is a constant. It can be declared and used as

```

int m = 1, n = 5;
int* const q = &m; // q is a const pointer, points to m
q = &n;            // error, constant q can not change

*q = n;           // ok, value that q points to is now n
int k = m;        // k = 5

```

Although  $q$  is a constant pointer that can only point to object  $m$ , the value of the object that  $q$  points to can be changed to the value of  $n$ , which is 5. Thus  $k$  is initialized to 5.

A related concept is a pointer that points to a constant object; that is, if  $p$  is such a pointer, then the value of the object pointed to by  $p$  can not be changed. It only says that  $*p$  can not be changed explicitly by using it as *lvalue*. However, the pointer  $p$  itself can be changed to hold the address of another object. It can be declared and used as

```
const int* p = &m;      // p points to constant object
*p = n;                // error, *p can not change explicitly
p = &n;                // ok, pointer itself can change
```

There is some subtlety involved here. Look at the example:

```
int m = 1, n = 5;
const int* p = &m;      // p points to m, so *p becomes 1
int i = *p;            // *p = m = 1, so i = 1
m = 3;                 // m = 3, so *p becomes 3
int j = *p;            // *p = 3, so j = 3
p = &n;                // ok, p itself can change, *p = 5
int k = *p;            // *p = n = 5, so k = 5
```

Since  $p$  points to  $m$  at first, the assignment  $m = 3$  changes  $*p$  to 3. Then the assignment  $p = \&n$  changes  $*p$  to the value of  $n$ , which is 5. In other words,  $*p$  has been changed implicitly.

To avoid the subtlety above, a *const* pointer that points to a *const* object can be declared:

```
int m = 1, n = 5;
const int* const r = &m;
    // r is a const pointer that points to a const value

int i = *r;            // i = 1, since *r = m = 1
r = &n;                // error, r is const pointer
*r = n;                // error, r points to const value

m = 3;                 // this is the only way to change *r
int j = *r;            // j = 3
```

Since  $r$  is a *const* pointer that points to a *const* value  $m$ , it can not be redefined to point to other objects, and  $*r$  can not be assigned to other values. The only way to change  $*r$  now is through changing  $m$ , the value of the object to which  $r$  points.

### 3.6.5 Void and Null Pointers

The *void pointer* (*void\**) points to an object of unknown type. A pointer of any type can be assigned to a variable of type *void\**, and two variables of type *void\** can be compared for equality and inequality. It can be declared and used as

```
void* pv;           // pv is a void pointer
int* pi;           // pi is int pointer
pv = pi;           // implicit conversion from int* to void*
```

Its primary use is to define functions that take arguments of arbitrary types or return an untyped object. A typical example is the C quicksort function *qsort()*, declared in *<stdlib.h>*, that can sort an array of elements of any type. See Exercise 3.14.24 or the book [Str97] for more details. However, templates (see Chapter 7) can be more easily and efficiently used to achieve the same goal.

The *null pointer* points to no object at all. It is a typed pointer (unlike the void pointer) and has the integer value 0. If at the time of declaration, a pointer can not be assigned to a meaningful initial value, it can be initialized to be the null pointer. For example,

```
double* dp = 0;    // dp initialized to 0, dp is null pointer
```

This statement is an initialization, which declares *dp* to be a pointer to *double* and initializes *dp* with the value 0. Since no object is located at address 0, *dp* does not point to any object. Later, *dp* can be assigned to hold the address of some object of type *double*:

```
double d = 55;
dp = &d;           // *dp = 55.0
*dp = 0;           // it causes d = 0.0, dp still points to d
```

Zero (0) is an integer. Due to standard conversions, 0 can be used as a constant of any integral, floating point, or pointer type. The type of zero is determined by context. No object is allocated with the address 0. Thus 0 can be used to indicate that a pointer does not refer to any object. The negation of a null pointer is *true*.

### 3.6.6 Pointers to Structures

A pointer can point to a structure. In this case, its members are accessed using the *->* operator. Space for structure objects can be allocated by the operator *new* and freed by *delete*. For example,

```
point2d ab = {'F', 3, -5}; // ab is of type point2d
point2d* p = &ab;         // let p point to ab
char name = p->nm;         // assign nm field of p to name
double xpos = p->x;        // assign x field of p to xpos
```

```

double ypos = p->y;           // assign y field of p to ypos

p->x = 15.0;                   // ab.x = 15
p->y = 26.0;                   // ab.y = 26
p->nm = 'h';                   // ab.nm = 'h'

point2d* q = new point2d;     // allocate space for q
q->x = 5;                      // assign value to its member
delete q;                     // deallocate space

```

where the structure *point2d* is defined in §3.4.

A structure can contain a member that points to the structure itself. For example,

```

struct stack {
    char* element;           // elements of stack are chars
    stack* next;             // points to next element in stack
};

```

This is allowed because *next* is only a pointer and occupies a fixed amount of space (*sizeof(int\*)* bytes). Such a self-referential structure is called a *dynamic data structure*; elements can be allocated and attached to it (or removed from it and deallocated) using the pointer *next*. C++ provides a standard library on stacks; see §10.1.4.

To define two or more structure types that refer to each other, a forward declaration is needed:

```

struct BB;                    // forward declaration, defined later

struct AA {                   // AA has a member pointing to BB
    char c;
    BB* b;                    // b is a pointer to BB
};

struct BB {                   // definition of structure BB
    AA* a;
    int i;
};

```

Without the forward declaration of *BB*, the use of *BB* in the declaration of *AA* would cause a syntax error. The name of a structure or other user-defined type can be used before the type is defined as long as the use does not require the size of the type or the names of its members to be known. This technique can be used to create many complicated data structures.

### 3.6.7 Pointers to Char

By a convention in C, a string constant is terminated by the null character `'\0'`, with value 0. Thus the size of `"hello"` is 6 and its type is *const char*[6] with the last element equal to `'\0'`. Due to its compatibility to C, C++ allows one to assign a string constant to *char\** directly. For example,

```
char* str = "hella"; // assign string constant to char*
str[4] = 'o';        // error, string constant can not change

char str2[] = "hella"; // array of 6 char, sizeof(str2) = 6
str2[4] = 'o';        // OK, now str2 = "hello"

char* str3 = new char [5];
str3[4] = 'o';        // OK
delete[] str3;
```

The C++ standard library `<string>` can be conveniently used for dealing with strings; see §4.5. C-style strings are better avoided if possible.

### 3.6.8 Pointers and Arrays

As in C, pointers and arrays are closely related. The name of an array can be used as a *const* pointer to its initial element. For example,

```
int v[5] = {6,9,4,5,7};
int* q = &v[0];        // point to initial element. *q = 6
int* p = v;            // point to initial element. *p = 6

int* r = &v[5];        // point to last-plus-one element
                        // but *r is undefined
int* s = v + 5;        // pointer arithmetic, *s is undefined
```

It is legal to declare a pointer to the last-plus-one element of an array. Since it does not point to any element of the array, the value that such a pointer points to is undefined. This is useful in writing many algorithms for a variety of data structures such as *vector* (§10.1.1) and *list* (§10.1.2), where *begin()* points to the initial element and *end()* points to the last-plus-one element; see the picture on Page 338.

## 3.7 References

A reference is an alternative name or alias for an object. For a type *T*, *T*& denotes a reference to *T*. A reference to an object *b* of type *T* is defined as

```
T& r = b;    // r is declared to be a reference to object b
```





All these declarations are called *prototypes*. It is only the types that matter in a prototype.

Every function that is called in a program must be defined somewhere and only once. The instructions that a function performs must be put inside a pair of braces. The *return* statement is used for a function to produce an output. A function definition also serves as its declaration. For example, the *square()* function can be defined as

```
int square(int x) {    // definition of a function
    return x*x;
}
```

A function contains a block of statements and may be called outside the block. After execution of a function, program control reverts to the statement immediately following the function call. For example,

```
int i = 5;
int j = square(i);    // call function square(i) with i = 5
int k = j;            // control comes here after call, k=25
```

However, when a function is called before its definition, the function declaration must appear before the statement that calls the function. Such a declaration is called a *forward declaration*. For example,

```
main() {
    int square(int);    // forward declaration
    int x = square(6); // call square() before its definition
}

int square(int x) {    // definition of function square()
    return x*x;
}
```

In the definition of the function *square()*, *x* is called a *parameter* (or *formal argument*) to the function. It does not take on any value until *square()* is called, when it is replaced by the *argument* (or *actual argument*) from the calling environment, as in *j = square(i)*. In this call, the parameter *x* is replaced by the argument *i*. The type of *i* is also checked against the type of *x* and some conversion may be performed when their types do not match.

The definition of a function can not appear inside another function. For example, the definition of *square()* above can not be put inside the *main()* function; it can be before or after *main()*, or in another file.

### 3.8.2 Function Overloading

We can define another *square()* function that takes a double and returns a double:

```
double square(double x) {    // definition of a function
    return x*x;
}
```

So can we define yet another taking a long double and returning a long double. These versions of the *square()* function can coexist in a program. The compiler can decide which version to use by comparing the type of the argument with the type of the parameter. Using the same name for similar operations on different types is called *overloading*.

Return type is not considered in overloading resolution. For example,

```
double a = square(5);        // call int square(int)
double b = square(5.0);      // call double square(double)
```

Neither are functions declared in different scopes. An example is:

```
int cubic(int);
double fg(int i) {
    double cubic(double);    // no overloading occurs here
    return cubic(i);         // call double cubic(double)
}
```

In call *cubic(i)*, the function *cubic(double)*, declared in the same scope, is used.

### 3.8.3 Argument Passing

There are two argument passing mechanisms in C++: pass by value and pass by reference. In pass by value, the argument is first evaluated at the time of the function call, and its value becomes the value of the parameter during the execution of the function. The default is pass by value, as in the following example.

```
int pass_val(int x) {        // default is pass by value
    x = x*x;
    return x + 5;
}

int i = 5;
int j = pass_val(i);
// now i = 5, j = 30
```

In the function call *j = pass\_val(i)*, the value of the argument *i*, which is 5, is passed to the parameter *x*. The function *pass\_val()* starts with the value *x = 5*, executes its block of statements, and returns value 30 to *j*. Thus after the function call, *i = 5* and *j = 30*.

In pass by reference, an argument must be a variable with allocated location and the argument is passed to the function as a reference, so that

the parameter becomes an alias to the argument and any change made to the parameter occurs to the argument as well. Pass by reference is signified by the reference operator `&`. For example, consider an example of pass by reference:

```
int pass_ref(int& x) {           // pass by reference
    x = x*x;
    return x + 5;
}

int i = 5;
int j = pass_ref(i);
// now i = 25, j = 30
```

In the function call  $j = \text{pass\_ref}(i)$ , the argument  $i$  is passed as a reference to the parameter  $x$ . That is, parameter  $x$  and argument  $i$  refer to the same object and  $x$  becomes an alias to  $i$ . Thus any change made to  $x$  inside the function also occurs to  $i$ . The function `pass_ref()` starts with the value  $x = 5$ , executes its block of statements, and returns value 30 to  $j$ . However, inside the function,  $x$  is updated to 25. This update occurs to  $i$  as well. Thus after the function call,  $i = 25$  and  $j = 30$ .

Pass by value for pointers can achieve the effect of pass by reference, since two pointers point to the same object if they have the same value. For example, compare the following three versions of the `swap()` function, which is intended to swap two numbers.

```
void swap(int& p, int& q) {      // pass by reference
    int tmp = p;                // a temporary variable
    p = q;                     // swap values of p and q
    q = tmp;
}

int i = 2, j = 3;
swap(i, j);
// now i = 3, j = 2

void swap(int* p, int* q) {     // pass by value for pointers
    int tmp = *p;
    *p = *q;
    *q = tmp;
}

i = 2, j = 3;
swap(&i, &j);                   // addresses of i, j are passed
// now i = 3, j = 2

void swap2(int p, int q) {      // pass by value
    int tmp = p;
```

```

    p = q;
    q = tmp;
}
i = 2, j = 3;
swap2(i, j);
// now i = 2, j = 3

```

In the first function call *swap()*, the references of *i* and *j* are passed to the function, and *p* and *q* become aliases to *i* and *j*, respectively. Thus any change made to *p* and *q* inside the function occur to *i* and *j* as well. In the second *swap()*, the values of *&i* and *&j*, that is, the addresses of *i* and *j*, are passed to the parameters *p* and *q*. Thus *p* points to object *i* and *q* points to object *j*, and swapping *\*p* and *\*q* is equivalent to swapping *i* and *j*. In the third version: *swap2()*, the values of *i* and *j* are passed to the parameters *p* and *q*, respectively, and they can not be changed by the function call. This version is useless since it does not swap the values of *i* and *j* at all.

Now look at another function that passes by value for its first argument and by reference for its second:

```

void f(int val, int& ref) {
    val++;
    ref++;
}
int i = 1, j = 1;
f(i, j);
// now j = 2, i = 1.

```

Pass by value does not change the value of the arguments and thus is safe. On the other hand, pass by reference or pass by value for pointers usually implies that the values of the arguments are going to be changed through the function call, unless they are explicitly told not to, using the keyword *const*. For example,

```

int g(int val, const int& ref) {
    // ref is not supposed to be changed
    val++;
    return ref + val;
}

```

Because of the *const* specifier, the compiler will give a warning if *ref* is to be changed inside the function. For example, it is illegal to write

```

void w(const int& ref) {
    ref = 5;           // WRONG, ref is not writable
}

```

Since the *const* specifier implies not to change the parameter *ref*, it can be substituted by a constant argument in a call:

```

g(5, 100);           // OK

f(5, 100);           // error, 2nd arg is not a variable
int i = 100;
f(5, i);             // OK, 2nd arg must be a variable

```

The name of an array is a *const* pointer that points to its initial element. Thus array arguments are treated as pointers and have the same effect of pass by reference. For example,

```

double h(double* const d, int n) { // d is a const pointer
    double sum = 0;
    for (int i = 0; i < n; i++) sum += d[i];
    d[n - 1] = 1000;               // but d[i] can change
    return sum;
}

double a[] = { 1, 2, 8, 20, -10, 30 };
double sum = h(a, 6);             // sum of elements in a
double d5 = a[5];                 // d5 = 1000

```

Constant pointers that point to constant objects or simply constant pointers can be used as safeguards to prevent accidental modification of certain variables that are not supposed to be modified. For example,

```

double fg(double* const d, int n) { // d is const pointer
    d = new double [n];             // error
    // ...
}

double gh(const double* const d, int n) {
    // d is const pointer that points to constant object
    double sum = 0;
    for (int i = 0; i < n; i++) sum += d[i];
    d[n - 1] = 1000;               // error, d points to constants
    return sum;                   // d[i] can not be changed
}

```

In the definition of *fg()*, the parameter *d* can not be made to point to another object since *d* is a *const* pointer. Using such a parameter indicates that space should not be allocated inside the function definition. Instead, the caller to this function should allocate space for it before calling. Similarly, the value of parameter *d* in function *gh()* should not be changed in the definition.

### 3.8.4 Return Values

A function that is not declared void must return a value, except for the function `main()`; see §3.8.10. For example,

```
int square(int& i) {
    i *= i;          // error, a value must be returned.
}
```

Each time a function is called, new copies of its arguments and *automatic variables* (like `tmp` in the `swap()` functions) are created. Since automatic variables will be destroyed upon the return of the function call, it is an error to return a pointer or reference to an *automatic variable*. The compiler normally gives a warning message so that such errors can be easily avoided. For example,

```
int* fa() {          // need to return address of int
    int local = 5;
    return &local;   // error
}                  // can not return address of local variable

int& fr() {          // need to return reference to int
    int local = 5;
    return local;    // error
}                  // can not return a reference to local var
```

However, it is not an error to return the value of a local pointer variable or the reference to a global variable. For example,

```
int* f() {
    int* local = new int;
    *local = 5;
    return local;    // return value of a local variable
}

int* i = f();
int j = *i;         // j = 5;
delete i;           // free space at address i allocated in f()

int v[30];
int& g(int i) {
    return v[i];     // it makes g(i) refer to v[i] (v is global)
}

g(3) = 7;           // now v[3] = 7, since g(3) refers to v[3]
```

Observe that objects created by operator *new* exist on the dynamic memory (heap, or free store). They still exist after the function in which they are created is returned. These objects can be explicitly destroyed by calling *delete*. The function `f()` returns the address of the object created by *new*

to the variable  $i$ . Thus  $*i$  gives the value of the object and the operator *delete* frees the object at address  $i$ . The function  $g()$  returns a reference to a global variable  $v$  and thus  $g(3)$  refers to  $v[3]$ .

For a void function, it is an error to return a value. However, an empty return statement can be used to terminate a function call, or a call to a void function can be made in a return statement. For example,

```
void h(int);

void hh(int& i) {
    int k = 0;
    for (int j = 0; j < 100; j++) {
        // do something to modify k and i
        if (k == 0) return;    // fcn terminates if k is zero
        if (k == 2) return h(i);
    }
    // call void function in return statement
}
```

As a more meaningful example of a function that returns a pointer, an  $n$  by  $m$  matrix multiplication with an  $m$ -column vector can be defined and used:

```
double* mxvrmy(const double** const mx,
               const double* const vr, int n, int m) {
    double* tmv = new double [n]; // tmv will store product
    for (int i = 0; i < n; i++) { // find the product
        tmv[i] = 0;
        for (int j = 0; j < m; j++) tmv[i] += mx[i][j]*vr[j];
    }
    return tmv; // return the address of the product
}

main() {
    int n = 100, m = 200;
    double** a = new double* [n]; // create space for matrix
    for (int i = 0; i < n; i++) a[i] = new double [m];
    double* b = new double [m]; // create space for vector

    for (int i = 0; i < n; i++) // assign values to matrix
        for (int j = 0; j < m; j++) a[i][j] = i*i + j;
    for (int j = 0; j < m; j++) b[j] = 3*j + 5;

    double* c = mxvrmy(a, b, n, m); // matrix-vector multiply
    double sum = 0; // find sum of elements
    for (int i = 0; i < n; i++) sum += c[i];
}
```



```

    for (int i = 0; i < n; i++) delete[] a[i];
    delete[] a;                      // free matrix a
    delete[] b;                      // free vector b
    delete[] c;                      // free vector c
}

```

The function *mxxrmy()* returns (the base address of) the product of the matrix *mx* with vector *vr*. Inside the *main()* program, some values are assigned to matrix *a* and vector *b*, and variable *c* is initialized to point to the initial element of the product vector, which is allocated and computed inside the function *mxxrmy()*. The summation of all elements in the product vector *c* is computed and stored in *sum*. Notice that the space for the product vector is allocated using *new* inside the function *mxxrmy()*, but is freed using *delete* in *main()*.

### 3.8.5 Recursive Functions

A function that calls itself in its definition is said to be a *recursive function*. For example, a function that calculates the factorial  $n!$  of a nonnegative integer  $n$  can be defined recursively:

```

long factorial(int n) {
    if (n == 0) return 1L;           // stopping statement
    return n*factorial(n-1);
}

```

Here *factorial(n)* is recursively defined as  $n * \text{factorial}(n - 1)$  and the recursion stops when it reaches *factorial(0)*. Thus a call like *factorial(2)* can be interpreted as

$$\text{factorial}(2) = 2 * \text{factorial}(1) = 2 * 1 * \text{factorial}(0) = 2 * 1 * 1 = 2.$$

Note that without the stopping statement: *if (n == 0) return 1L*, the program would cause an infinite recursion.

Recursive functions may impose a lot of run-time overhead compared to nonrecursive functions. For example, a recursive function call *factorial(2)* requires actually three function calls (*factorial(2)*, *factorial(1)*, and *factorial(0)*). The overhead of function calls can significantly slow down a program when there are a large number of them.

### 3.8.6 Inline Functions

When a function is called, its arguments are copied and the program branches out to a new location (on the run-time stack; see §3.9) and comes back after the instructions contained in the function are executed. To avoid

such function calling overhead, *inline* functions can be defined using the keyword *inline* for small functions. Statements in an inline function are substituted in the program calling it during compilation. For example, an inline function can be defined as

```
inline int twice (int i) { return 2*i; }
```

If somewhere in the program there is a statement like  $y = \text{twice}(x)$ ; then the content of  $\text{twice}(x)$  will be substituted. That is, the compiler will generate the statement  $y = 2 * x$  and a run-time function call will be avoided.

This can also be achieved as normally done in C by a macro:

```
#define twice(i) (2*(i))
```

Then every occurrence of  $\text{twice}(x)$  will be substituted by  $(2 * (x))$  in the compiling process. Compared to C-style macros, inline functions are type-checked and safer to use.

However, the *inline* specifier is only a hint to the compiler that it should attempt to generate code substitution. The possibility of mutually recursive inline functions and other situations make it impossible to guarantee that every call of an *inline* function is actually inlined. When it can not be inlined, the usual function call mechanism will be used and the keyword *inline* will be ignored.

### 3.8.7 Default Arguments

A function can have *default arguments* for its trailing arguments. A default value for an argument can be specified in a function definition. For example,

```
void print(int v, int b = 10) { // b is defaulted to be 10
    // define the function to print out integer v in base b
}
```

```
print(15,16);    // print 15 in hexadecimal (base 16)
print(15,2);     // print 15 in binary (base 2)
print(15,10);    // print 15 in decimal (base 10)
print(15);       // print 15 in decimal (base 10)
```

The last call is equivalent to  $\text{print}(15,10)$ . That is, when a value is not explicitly given for the second argument, 10 is assumed by default.

The default values for arguments can also be specified in a function declaration. For example,

```
void print(int = 100, int = 10);    // 2 default arguments

print(15,16);    // print 15 in hexadecimal (base 16)
print(15,2);     // print 15 in binary (base 2)
print(15);       // print 15 in decimal (base 10)
```

```

print();           // print 100 in decimal (base 10)

void print(int v, int b) {
    // define the function to print out integer v in base b
}

```

Note that default arguments are possible only for trailing arguments. The compiler will be confused at

```

void print(int b = 10, int);           // error

```

### 3.8.8 Function Types

It is sometimes useful to define a type synonym using the keyword *typedef*. A declaration prefixed by *typedef* introduces a new name for a type rather than declaring a new variable. For example,

```

typedef unsigned int Uint;           // Uint is unsigned int
Uint i = 5;                          // i is an unsigned int

typedef double (*coef)(double);
                        // coef means a pointer to function
coef f;                       // f has type of coef

```

A *typedef* does not introduce a new type. For example, *Uint* and *unsigned int* are the same type, and *coef* is the same type as a pointer to a function that takes a *double* and returns a *double*. The last statement above declares *f* to be a variable of type *coef*. That is, *f* can take on values that are pointers to a function that takes a *double* and returns a *double*.

A pointer to a function can be passed as an argument to another function like any other pointer. For example,

```

double integral(coef f, double a, double b){
    return f((a+b)*0.5)*(b-a); // an inaccurate evaluation
}                               // of a definite integral

double square(double d) { return d*d; }

coef sfp = square;           // initialize sfp with 'square()'
double v = integral(sfp, 2, 2.01); // call integral()
sfp = sqrt;                  // assign sqrt in <math.h> to sfp
v = integral(sfp, 2, 2.01);   // call integral()

```

The function call *integral(square, 2, 2.01)* now gives an approximate value of the definite integral of the function *square()* from 2 to 2.01. The name of a function is treated as a pointer to the function.

Here is a more complicated example from a C header file *<signal.h>*:

```
typedef void (*SIG_TYP) (int);      // from <signal.h>
typedef void (*SIG_ARG_TYP) (int); // pointer-to-fcn type
SIG_TYP signal(int, SIG_ARG_TYP); // return pointer to fcn
```

This defines *SIG\_TYP* and *SIG\_ARG\_TYP* to be the type of pointer to functions that take an *int* as argument and do not return a value, and *signal* is a function returning a value of type *SIG\_TYP* and taking an *int* and a *SIG\_ARG\_TYP* as arguments.

### 3.8.9 Static Local Variables

Variables defined within the body of a function are automatic variables by default. They lose their value after the function call is completed. This property can be overridden by use of the keyword *static*. A *static variable*, although still local, persists across function invocations and maintains its value after the function call is completed. Consider a function that calculates a value only when it is first called and just returns the value without repeating the calculation in subsequent calls:

```
long double cal() {
    static bool first = true;
    static long double v;
    if (first == true) {
        v = 5.0;          // or perform some calculation to get v
        first = false;
    }
    return v;
}

long double x = cal();
           // 1st call, the 'if' block is executed in cal()
long double y = cal();
           // 2nd call, the 'if' block is skipped in cal()
```

The value of the variable *first* is initialized to *true* when the function *cal()* is first called. Then it is assigned to have a new value *false* inside the function. This new value will be retained and the initialization statement

```
static bool first = true;
```

will be ignored in subsequent calls. Since *first = false*, the *if* statement in *cal()* is then skipped. The same explanation holds for the *static* local variable *v*. For this reason, assignment *v = 5.0* is executed only in the first call.

The value of a static variable can also change with each function call. For example,

```
int gcd(int m, int n) {          // greatest common divisor
```

```

static int callnum = 1;
cout << "gcd() has been called " << callnum++
    << " times so far.\n";

if (n == 0) return m;
return gcd(n,m%n);
}

```

When `gcd()` is first called, `callnum = 1` and `callnum` is then given a new value 2. This value will be maintained when the second time `gcd()` is called and the initialization `int callnum = 1` will be ignored. In the second call, `callnum` is incremented by 1 to a new value 3. With the static variable `callnum`, one can know how many times a function is called in a program. For example, in calculating the greatest common divisor `gcd(215, 335)` the recursive function need be called eight times. However, look at the following two calls of the function.

```

int i = gcd(215,335);    // after this call, callnum = 8
int j = gcd(215,335);    // after this call, callnum = 16

```

At the beginning of the second call `j = gcd(215, 335)`, the value of `callnum` is 9 (from the first call `i = gcd(215, 335)`). The value of `callnum` will be incremented from 9 to 16 during the second call, since the static variable `callnum` persists throughout the program.

A static variable provides a function with “memory” without introducing a global variable that might be accidentally accessed and corrupted by other functions.

### 3.8.10 The Function *main*

The minimal C++ program is

```
int main() { }
```

It takes no arguments and does nothing. Every C++ program must have a function called `main()`. The program starts by executing this function. The `int` value returned to the system indicates the program has completed successfully or otherwise. If no value is explicitly returned, the system will receive a zero value for successful completion and nonzero value for failure.

The function `main()` may take two arguments specifying the number of arguments, usually called `argc`, and the array of arguments, usually called `argv`, for the program. The arguments are character strings having type `char* [argc + 1]`. The name of the program is passed as `argv[0]` and the list of arguments is zero-terminated; that is, `argv[argc] = 0`.

For example, consider the simple program in §1.1. Instead of inputting two integers after the program is run, we want to input them at the UNIX/Linux command line as

```
add 1 1000
```

Here *add* is the name of the executable object code for the program. Then *add* is stored in *argv*[0], and 1 and 1000 will be stored in *argv*[1] and *argv*[2], respectively. The number of arguments is *argc* = 3. Note that *argv*[3] is 0 indicating the end of arguments. Now we need to modify the program to support this input format:

```
#include <iostream>
#include <stdlib.h>      // library for function atoi()
int main(int argc, char* argv[]) {
    if (argc != 3)
        cout << "number of arguments is incorrect.\n";

    int n = atoi(argv[1]); // first input int assigned to n
    int m = atoi(argv[2]); // 2nd input int assigned to m

    if (n > m) {           // if n > m, swap them
        int temp = n;      // declare temp, initialize it
        n = m;             // assign value of m to n
        m = temp;          // assign value of temp to m
    }

    double sum = 0;        // sum initialized to 0
    for (int i = n; i <= m; i++) sum += i;
    cout << "Sum of integers from " << n << " to " << m
         << " is: " << sum << '\n'; // output sum to screen
}
```

Note that the function *atoi*() in the standard header *<stdlib.h>* converts a character string into an integer. A related function *atof*() converts a character string into a float.

### 3.9 Program Execution

When a program is run, the execution is typically arranged as shown in Table 3.1. It consists of several portions.

- The *text* portion contains the actual machine instructions that are executable by the hardware. These machine instructions are generated by the compiler from the source code of the program. When a program is executed by the operating system, the text portion is typically read into memory from its disk file.
- The *data* portion contains the program's data, which can be further divided into *initialized data* (initialized by the program) and *unini-*

text	initialized data	uninitialized data	<div> <div>⋮</div> <div>heap ⋮ → ← ⋮ stack</div> <div>⋮</div> </div>
------	---------------------	-----------------------	--

TABLE 3.1. A typical arrangement of a program execution.

*tialized data*. Uninitialized data contain items that are not initialized by the program but are set to a default value (typically zero) before the program starts execution. A program file consists of the program text and initialized data. The advantage of providing an uninitialized data area is that the system does not have to allocate space in the program file on the disk for uninitialized global variables. It also reduces the time to read a program file from disk into memory.

- The *heap* is used to allocate more data space dynamically for the program while it is being executed.
- The *stack* is used dynamically to contain the stack frames that include the return linkage for each function call and also the data elements required by the function.

A gap between the heap and the stack indicates that many operating systems leave some room between these two portions so that they can grow dynamically during a program execution. When there is no more room for these two portions to grow, a run-time error will occur.

As an example, consider the following short program:

```

const int param = 10000;
double myvalue;

double f() {
    double* a = new double [param];
    for (int i = 0; i < param; i++) a[i] = rand() + i;

    double sum = 0;
    for (int i = 0; i < param; i++) sum += a[i];
    return sum;
}

int main() {
    myvalue = f();
    double s = sqrt(myvalue);
    cout << "result is: " << s + 555 << '\n';
}

```

The variable *param* is stored in the initialized data area and *myvalue* in uninitialized data. The constants 10000, 555, and “*result is:* ” (string literal) can be stored as (read-only) initialized data. The variables *a*, *i*, and *sum* are automatic variables and will be stored on the stack. When the function *f()* is called in the statement *myvalue = f()*, the system creates the objects *a*, *i*, and *sum* on the stack and 10000 objects pointed to by *a* on the heap. After the computation in the function call is finished, the objects *a*, *i*, and *sum* are destroyed and the computed value of *sum* is returned to the variable *myvalue*. However, the 10000 objects created on the heap are not explicitly deleted and thus still exist throughout the rest of the program execution. The automatic variable *s* is also stored on the stack while the function *main()* is being executed. The machine instructions for the functions *main()*, *f()*, *sqrt()*, and *rand()* are in the text segment.

### 3.10 Operator Summary and Precedence 优先

This section presents a summary of operators and their precedence, some of which are explained later in this book. Each operator is followed by its name and an example illustrating its use. Tables 3.2 and 3.3 contain a summary of all C++ operators and their precedence, where a *class\_name* is the name of a class, structure, or union, and a *member* or *Mem* is the name of one of its members. An *object* is an expression yielding an object, a *pointer* an expression yielding a pointer, an *expr* or *Er* is an expression, an *lvalue* is an expression denoting a nonconstant object, and a *type* or *Tp* can be a basic type or a derived type (with \*, (), etc.) with possible restrictions.

In Tables 3.2 and 3.3, each box holds operators with the same precedence. Operators presented in a higher box have higher precedence than operators in lower boxes. For example, the operator \* has higher precedence than – and  $a - b * c$  means  $a - (b * c)$  instead of  $(a - b) * c$ . Similarly,  $i <= 0 || 100 < i$  means  $(i <= 0) || (100 < i)$ , instead of the legal but nonsense expression  $i <= (0 || 100 < i)$ . Note that there are cases when the operator precedence does not result in obvious interpretation, such as

```
if (i & mask == 0 ) { /* do something */ }
```

Because of the higher precedence of == over the ‘bitwise and’ operator &, the expression is interpreted as  $i \& (mask == 0)$ . In this case, parentheses should be used to yield the desired result:

```
if ( (i & mask) == 0 ) { /* do something */ }
```



Operator Summary and Precedence		
Operator	Name	Example
::	scope resolution	class_name :: Mem
::	scope resolution	namespace_name :: Mem
::	global operator	:: name
.	member selection	object.member
->	member selection	pointer->member
[]	subscripting	pointer[expr], array[expr]
()	function call	expr(expr_list)
()	value construction	type(expr_list)
++	postincrement	lvalue++
--	postdecrement	lvalue--
()	type identification	typeid(type)
()	type identification	typeid(expr)
<i>dynamic_cast</i>	run-time cast	<i>dynamic_cast</i> <Tp>(Er)
<i>static_cast</i>	compile-time cast	<i>static_cast</i> <Tp>(Er)
<i>reinterpret_cast</i>	unchecked cast	<i>reinterpret_cast</i> <Tp>(Er)
<i>const_cast</i>	<i>const</i> cast	<i>const_cast</i> <Tp>(Er)
<i>sizeof</i>	size of object	<i>sizeof</i> (expr)
<i>sizeof</i>	size of type	<i>sizeof</i> (type)
++	preincrement	++lvalue
--	predecrement	--lvalue
~	bitwise complement	~ expr
!	not (negation)	!expr
-	unary minus	- expr
+	unary plus	+ expr
&	address of	&lvalue
*	dereference	*expr
<i>new</i>	allocate one object	<i>new</i> type
<i>new</i>	allocate objects	<i>new</i> type [expr]
<i>new</i>	allocate, initialize	<i>new</i> type (expr)
<i>new</i>	create (place an object)	<i>new</i> (expr) type
<i>new</i>	create (place objects)	<i>new</i> (expr) type [expr]
<i>new</i>	create (place, initialize)	<i>new</i> (expr) type (expr)
<i>delete</i>	deallocate one object	<i>delete</i> pointer
<i>delete</i>	deallocate objects	<i>delete</i> [] pointer
()	cast (type conversion)	(type) expr
.*	member selection	object.*pointer_to_Mem
->	member selection	pointer->pointer_to_Mem
*	multiply	expr * expr
/	divide	expr / expr
%	modulo (remainder)	expr % expr

TABLE 3.2. Operator summary and precedence.

Operator Summary and Precedence (continued)		
Operator	Name	Example
+	add (plus)	expr + expr
-	subtract (minus)	expr - expr
<<	shift left	expr << expr
>>	shift right	expr >> expr
<	less than	expr < expr
<=	less than or equal to	expr <= expr
>	greater than	expr > expr
>=	greater than or equal to	expr >= expr
==	equal	expr == expr
!=	not equal	expr != expr
&	bitwise AND	expr & expr
^	bitwise exclusive OR	expr ^ expr
	bitwise inclusive OR	expr   expr
&&	logic AND	expr && expr
	logic inclusive OR	expr    expr
=	simple assignment	lvalue = expr
*=	multiply assignment	lvalue *= expr
/=	divide assignment	lvalue /= expr
%=	modulo assignment	lvalue %= expr
+=	add assignment	lvalue += expr
-=	subtract assignment	lvalue -= expr
<<=	shift left assignment	lvalue <<= expr
>>=	shift right assignment	lvalue >>= expr
&=	AND assignment	lvalue &= expr
^=	exclusive OR assignment	lvalue ^= expr
=	inclusive OR assignment	lvalue  = expr
? :	conditional expression	expr ? expr: expr
<i>throw</i>	throw exception	<i>throw</i> expr
,	comma (sequencing)	expr, expr

TABLE 3.3. Continuation of operator summary and precedence.

Another example is:

```
if (0 > i || i > 100 && abs(i) < 500) { /* ... */ }
```

which is equivalent to

```
if (0 > i || (i > 100 && abs(i) < 500)) { /* ... */ }
```

where the function *abs(i)* gives the absolute value of an integer *i* (declared in the standard header *<stdlib.h>*). Note that it is different from

```
if ((0 > i || i > 100) && abs(i) < 500) { /* ... */ }
```

This can be easily checked by taking  $i = -1000$  in the two statements. Also, the mathematical expression  $0 \leq x < 100$  can be translated into the C++ expression

```
0 <= x && x < 100
```

or equivalently  $(0 \leq x) \&\& (x < 100)$ , but never

```
0 <= x < 100
```

When in doubt, parentheses should be used; it may also increase readability of the code.

All operators are left associative, except for unary operators and assignment operators, which are right associative. For example,  $a - b - c$  means  $(a - b) - c$  and  $a || b || c$  means  $(a || b) || c$ , but  $a = b = c$  means  $a = (b = c)$  and  $*p++$  means  $*(p++)$  rather than  $(*p)++$  (i.e.,  $q = *p++$  means  $q = *p$ ;  $p++$ );).

A few grammar rules can not be expressed in terms of operator precedence and associativity. For example, the expression  $a = b < c ? d = e : f = g$  means  $a = ((b < c) ? (d = e) : (f = g))$ .

The order of evaluation of subexpressions in an expression is undefined in general. For example, in the statement

```
int i = f(x) + g(y);    // undefined order of evaluation
```

it is undefined whether  $f(x)$  or  $g(y)$  is called first; and the result of

```
int i = 5;
v[i] = i++;              // undefined results
```

is unpredictable (it may be  $v[5] = 5$  or  $v[6] = 5$  depending on whether the left-hand operand or right-hand operand is evaluated first). A program should not assume any order of evaluation for such expressions.

However, the three operators `,` (comma), `&&` (logical and), and `||` (logical or) guarantee that their left-hand operand is evaluated before the right-hand operand. See §2.2.5 and §2.2.3.

The following keywords

<b>and</b>	<b>and_eq</b>	<b>bitand</b>	<b>bitor</b>	<b>compl</b>	<b>not</b>
<b>not_eq</b>	<b>or</b>	<b>or_eq</b>	<b>xor</b>	<b>xor_eq</b>	

are provided for some non-ASCII character sets, where characters such as & and ^ are not available. They are equivalent to the following tokens, respectively.

&&	&=	&		~	!
!=		=	^	^=	

However, they can also be used even with ASCII character sets for clarity. For example, *not\_eq* may be more readable than != to some people.

### 3.11 Standard Library on Mathematical Functions

The standard library `<math.h>` provides the following mathematical functions.

```
double abs(double);
    // absolute value, not in C; same as C's fabs()
double fabs(double);
    // absolute value, inherited from C

double ceil(double d);
    // smallest integral value not less than d
double floor(double d);
    // largest integer not greater than d

double sqrt(double d);
    // square root of d, d must be nonnegative
double pow(double d, double e);    // d to the power of e
    // error if d ==0 and e<= 0
    // and error if d < 0 and e is not an int
double pow(double d, int i);
    // d to the power of i, not in C

double cos(double);                // cosine
double sin(double);                // sine
double tan(double);                // tangent

double acos(double);               // arc cosine
double asin(double);               // arc sine
double atan(double);               // arc tangent
double atan2(double x, double y); // atan(x/y)
```

```

double cosh(double);    // hyperbolic cosine
double sinh(double);    // hyperbolic sine
double tanh(double);    // hyperbolic tangent

double acosh(double);   // inverse hyperbolic cosine
double asinh(double);   // inverse hyperbolic sine
double atanh(double);   // inverse hyperbolic tangent

double exp(double);     // natural exponential, base e
double log(double d);   // natural (base e) logarithm
                        // d must be > 0
double log2(double d);  // base 10 logarithm, d must be > 0

double modf(double d, double* p);
                        // return fractional part of d, and store
                        // integral part of d in *p
double fmod(double d, double m);
                        // floating point remainder, same sign as d
double ldexp(double d, int i);    // d*exp(2,i)
double frexp(double d, int* p);
                        // return mantissa of d, put exponent in *p

```

In addition, `<math.h>` supplies these functions for *float* and *long double* arguments. For complex arguments with real and imaginary parts in *float*, *double*, and *long double* precisions, see §7.4. The angle returned by the inverse trigonometric functions `asin()`, `acos()`, `atan()`, and `atan2()` are in radians. The range of `asin()` and `atan()` is  $[-\pi/2, \pi/2]$ . The range of `acos()` is  $[0, \pi]$ . The range of `atan2()` is  $[-\pi, \pi]$ , whose principal use is to convert Cartesian coordinates into polar coordinates. For `asin()` and `acos()`, a domain error occurs if the argument is not in  $[-1, 1]$ .

The function `frexp(double d, int* p)` splits a value *d* into mantissa *x* and exponent *p* such that  $d = x \times 2^p$ , and returns *x*, where the magnitude of *x* is in the interval  $[0.5, 1)$  or *x* is zero. The function `fmod(double d, double m)` returns a floating modulus  $d \bmod m$ . If *m* is nonzero, the value  $d - i \times m$  is returned, where *i* is an integer such that the result is zero, or has the same sign as *d* and magnitude less than the magnitude of *m*. If *m* is zero, the return value is system-dependent but typically zero.

Errors are reported by setting the variable `errno` in the standard library `<errno.h>` to `EDOM` for a domain error and to `ERANGE` for a range error, where `EDOM` and `ERANGE` are macros defined in `<errno.h>`. For example,

```

int main() {
    errno = 0;    // clear old error state, include <errno.h>
    double x = - 50;    // x can also be computed at run-time

```

```

double i = sqrt(x);
if (errno == EDOM)
    cerr << "sqrt() not defined for negative numbers\n";
double p = pow(numeric_limits<double>::max(), 2);
if (errno == ERANGE)
    cerr << "result of pow(double,int) overflows\n";
}

```

where *cerr* represents the standard error stream (usually the terminal screen), defined in *<iostream>*. Thus, by checking the value of the variable *errno*, the user can know if a domain or range error has occurred in calling standard functions in *<math.h>*.

For historical reasons, a few mathematical functions are in *<stdlib.h>* rather than *<math.h>*:

```

int abs(int);           // absolute value
long abs(long);        // absolute value, not in C
long labs(long);       // absolute value

struct div_t { implementation_defined quot, rem};
struct ldiv_t { implementation_defined quot, rem};

div_t div(int n, int d);
    // divide n by d, return (quotient, remainder)
ldiv_t div(long int n, long int d);
    // divide n by d, return (quotient, remainder), not in C
ldiv_t ldiv(long int n, long int d);
    // divide n by d, return (quotient, remainder)

double atof(const char* p); // convert p to double
int atoi(const char* p);    // convert p to int
long atol(const char* p);   // convert p to long int

```

In the definition of the structures *div\_t* and *ldiv\_t*, the type of *quot* and *rem* is implementation-dependent. The functions *atof*, *atoi*, and *atol* convert strings representing numeric values into numeric values, with the leading whitespace ignored. If the string does not represent a number, zero is returned. For example, *atoi("seven")* = 0, *atoi("77")* = 77, while *atoi('s')* gives the numeric value representing the lowercase letter s.

### 3.12 Polynomial Evaluation

In many scientific applications, the value of an *n*th degree polynomial

$$p_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

needs to be evaluated for given values of  $x$ , where the coefficients  $a_n, a_{n-1}, \dots, a_1, a_0$  are known. A C++ function can be written to do this job having prototype

```
double eval(double* a, int n, double x);
```

where  $a$  is a pointer that points to the coefficients  $a_0, a_1, \dots, a_{n-1}, a_n$ . An inner *for* loop may be used to do the multiplication  $a_k x^k$  and an outer *for* loop for adding all the terms. Thus the function  $eval()$  can be straightforwardly defined as

```
double eval(double* a, int n, double x) {
    double sum = 0;
    for (int k = n; k >= 0; k--) {
        double xpowerk = 1;
        for (int i = 1; i <= k; i++) {    // or call pow(x, k)
            xpowerk *= x;                // compute x to power of k
        }
        sum += a[k]*xpowerk;            // add each term to sum
    }
    return sum;
}
```

Notice that evaluating the first term  $a_n x^n$  requires  $n$  multiplications and the second term  $a_{n-1} x^{n-1}$  requires  $n-1$  multiplications, and so on. Thus the total number of multiplications to evaluate  $p_n(x)$  using the function call  $eval(x)$  is  $n + (n-1) + \dots + 2 + 1 = n(n+1)/2$ . This method is very inefficient since it duplicates the work in computing the power of  $x$  for each term of  $p_n(x)$ .

An efficient method called Horner's algorithm (or nested multiplication) is described below. The idea is to rewrite  $p_n(x)$  in a nested multiplication form:

$$p_n(x) = (\dots((a_n x + a_{n-1})x + a_{n-2})x + \dots + a_1)x + a_0.$$

An example of writing a polynomial of degree 4 from its regular form into nested multiplication form is:

$$4x^4 + 3x^3 + 2x^2 + x + 10 = (((4x + 3)x + 2)x + 1)x + 10.$$

Introduce the notation:

$$\begin{aligned} u_n &= a_n; \\ u_{n-1} &= u_n x + a_{n-1}; \\ u_{n-2} &= u_{n-1} x + a_{n-2}; \\ &\vdots \\ u_1 &= u_2 x + a_1; \\ u_0 &= u_1 x + a_0; \end{aligned}$$

Thus  $p_n(x) = u_0$ . At first glance,  $u_n, u_{n-1}, \dots$ , and  $u_0$  would be stored in an array. However, it is not necessary since after computing  $u_{n-1}$  the value  $u_n$  is no longer needed, and so on. This procedure can be written into an algorithm.

**Algorithm 3.12.1** *horner*( $a, n, x$ )

- Initialize  $u = a_n$ .
- for ( $i = n - 1, n - 2, \dots, 0$ ) {
  - $u \leftarrow ux + a_i$ ;
- return  $u$ .

It takes only  $n$  multiplications. Now Horner's algorithm can be coded into C++ as

```
double horner(double* a, int n, double x) {
    double u = a[n];
    for (int i = n - 1; i >= 0; i--) u = u*x + a[i];
    return u;
}
```

This program is not only shorter, but also more efficient than the straightforward approach *eval*(). Efficient algorithms require fewer operations and may sometimes result in smaller accumulative roundoff errors.

Now these two functions can be compared by evaluating an 8th degree polynomial:

$$p_8(x) = x^8 - 8x^7 + 28x^6 - 56x^5 + 70x^4 - 56x^3 + 28x^2 - 8x + 1$$

for  $x$  taking on 11 equally spaced values covering interval  $[0.99999, 1.0001]$ . Observe that  $p_8(x) = (x - 1)^8$ . This expression will be compared to the values returned by *eval*() and *horner*().

```
#include <iostream>
#include <math.h>
int main() {
    double a[9] = {1, - 8, 28, - 56, 70, - 56, 28, - 8, 1};
    for (double x = 0.99999; x <= 1.0001; x += 1.0e-5) {
        cout.width(18);
        cout << eval(a,9,x);    // straightforward evaluation
        cout.width(18);
        cout << horner(a,9,x);  // Horner's algorithm
        cout.width(18);
        cout << pow(x-1,8) << '\n';    // direct evaluation
    }
}
```



The output on a computer with 8 bytes for *double* is as follows.

5.32907e-15	2.14564e-16	1e-40
0	0	0
3.55271e-15	1.32321e-15	1e-40
0	-1.59367e-15	2.56e-38
0	-1.76064e-15	6.561e-37
-2.66454e-15	-6.9931e-17	6.5536e-36
4.44089e-15	-3.5203e-15	3.90625e-35
-6.21725e-15	-3.23331e-15	1.67962e-34
-5.32907e-15	-1.87686e-15	5.7648e-34
8.88178e-15	-2.31586e-16	1.67772e-33
2.66454e-15	5.89264e-16	4.30467e-33

The correct value for  $p_8(x) = (x - 1)^8$  should be nonnegative and very small since  $x$  is close to 1. The third column is the best since it is a direct calculation of  $(x - 1)^8$  and will be used as exact values in the comparison. Comparing the first column (produced by *eval()*) with the second column (produced by *horner()*), the results by Horner's algorithm are more accurate (closer to the third column) for 8 out of 11 values. Both *eval()* and *horner()* produce negative values due to the subtractions of nearly equal quantities in their evaluations.

Let us now enlarge the errors by  $10^{10}$  times in the evaluations of *eval()* and *horner()* so that their accuracies can be more clearly seen. Modify the main program as

```
int main() {
    double a[9] = {1, - 8, 28, - 56, 70, - 56, 28, - 8, 1};
    for (double x = 0.99999; x <= 1.0001; x += 1.0e-5) {
        cout.width(20);
        cout << 1.0e10*(pow(x-1,8) - eval(a,9,x));
        cout.width(20);
        cout << 1.0e10*(pow(x-1,8) - horner(a,9,x)) << '\n';
    }
}
```

The output becomes:

-5.32907e-05	-2.14564e-06
0	0
-3.55271e-05	-1.32321e-05
2.56e-28	1.59367e-05
6.561e-27	1.76064e-05
2.66454e-05	6.9931e-07
-4.44089e-05	3.5203e-05
6.21725e-05	3.23331e-05
5.32907e-05	1.87686e-05

-8.88178e-05	2.31586e-06
-2.66454e-05	-5.89264e-06

For 9 out of 11 values, the second column is no larger than the first.

In large-scale applications, it is not enough to compute correctly, but also efficiently and elegantly. The elegance of a program should include extensibility, maintainability, and readability, while correctness should also include accuracy, robustness, and portability. Both *horner()* and *eval()* are correct (*horner()* is slightly more accurate in certain cases), but the efficiency of *horner()* is far better for polynomials of high degrees. In finance, evaluation of high-degree polynomials is often encountered; see Exercises 3.14.18 and 4.8.12.

### 3.13 Trapezoidal and Simpson's Rules

In this section, a complete example is presented to evaluate definite integrals of the form  $\int_a^b f(x)dx$  using the Trapezoidal Rule and Simpson's Rule, based on partitioning the interval  $[a, b]$  into  $n$  subintervals, where  $n$  is a positive integer. We define two functions *trapezoidal()* and *simpson()* that take the integral bounds  $a$  and  $b$ , the integrand  $f$ , and the number of subintervals  $n$  as arguments, and return a value for the approximate integral.

We want to write a main program as

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

typedef double (*pfn)(double); // define pfn for integrand
double trapezoidal(double a, double b, pfn f, int n);
double simpson(double a, double b, pfn f, int n);

double square(double d) { return d*d; }

int main(){
    double result = trapezoidal(0, 5, square, 100);
    cout << "Integral using trapezoidal with n = 100 is: "
         << result << '\n';

    result = simpson(0, 5, square, 100);
    cout << "Integral using simpson with n = 100 is: "
         << result << '\n';

    result = trapezoidal(0, 5, sqrt, 100);
    cout << "Integral using trapezoidal with n = 100 is: "
```

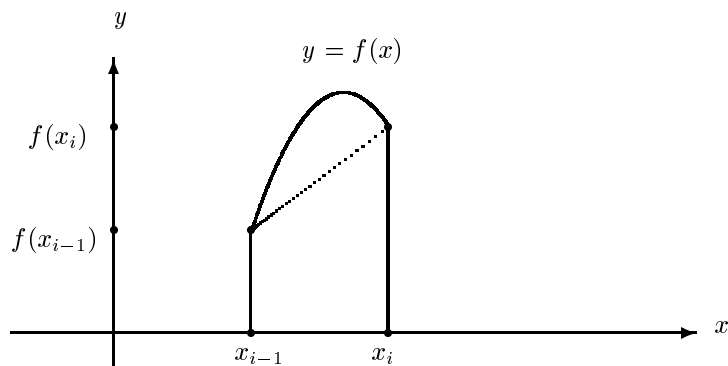


FIGURE 3.2. The integral  $\int_{x_{i-1}}^{x_i} f(x)dx$ , representing the area of the region under curve  $y = f(x)$  and above subinterval  $[x_{i-1}, x_i]$ , is approximated by the area of a trapezoid (with one edge drawn in dotted line), which is  $[f(x_{i-1}) + f(x_i)]h/2$ .

```

    << result << '\n';

    result = simpson(0, 5, sqrt, 100);
    cout << "Integral using simpson with n = 100 is: "
        << result << '\n';
}

```

Here *sqr*t() is the square-root function declared in *<math.h>*. This main program intends to find approximate values for  $\int_0^5 x^2 dx$  and  $\int_0^5 \sqrt{x} dx$  using the Trapezoidal and Simpson's Rules. Since these integrals can be evaluated exactly, they can be used as examples for testing the correctness of the program. An easy modification of the program enables one to find more complicated integrals such as  $\int_0^1 \sin(x^2) dx$  and  $\int_0^1 \sqrt{1 - \sin^3 x} dx$ .

Now we need to define the functions for the Trapezoidal and Simpson's Rules. The idea of the Trapezoidal Rule is to first partition the given interval  $[a, b]$  into  $n$  (assuming equally spaced for simplicity) subintervals  $[x_0, x_1], [x_1, x_2], \dots, [x_{n-1}, x_n]$  with  $x_i - x_{i-1} = h$ , where  $h$  is the size of each subinterval. Then approximate the integral  $\int_{x_{i-1}}^{x_i} f(x)dx$  on each subinterval  $[x_{i-1}, x_i]$  by the area of a trapezoid passing through four points  $(x_{i-1}, 0)$ ,  $(x_{i-1}, f(x_{i-1}))$ ,  $(x_i, f(x_i))$ , and  $(x_i, 0)$  (assuming  $f(x_{i-1})$  and  $f(x_i)$  are positive, but the result also applies to general functions). The area of such a trapezoid is  $(f(x_{i-1}) + f(x_i)) * h/2$ . See Figure 3.2.

Thus,

$$\int_a^b f(x)dx = \sum_{i=1}^n \int_{x_{i-1}}^{x_i} f(x)dx$$

$$\approx \sum_{i=1}^n \frac{f(x_{i-1}) + f(x_i)}{2} h = \left[ \frac{f(x_0)}{2} + \sum_{i=1}^{n-1} f(x_i) + \frac{f(x_n)}{2} \right] h,$$

where the subinterval size  $h = (b - a)/n$  and partitioning points are  $x_i = a + i * h, i = 0, 1, \dots, n$ , with  $x_0 = a$  and  $x_n = b$ . This function can be defined as

```
double trapezoidal(double a, double b, pfn f, int n) {
    double h = (b - a)/n;          // size of each subinterval
    double sum = f(a)*0.5;
    for (int i = 1; i < n; i++) sum += f(a + i*h);
    sum += f(b)*0.5;
    return sum*h;
}
```

Simpson's Rule is similar to the Trapezoidal Rule but approximates the integral  $\int_{x_{i-1}}^{x_i} f(x)dx$  on each subinterval  $[x_{i-1}, x_i]$  by the area under a quadratic curve (represented by a polynomial of degree two) passing through three points  $(x_{i-1}, f(x_{i-1}))$ ,  $(\bar{x}_i, f(\bar{x}_i))$ , and  $(x_i, f(x_i))$ , where  $\bar{x}_i = (x_{i-1} + x_i)/2$  is the middle point of the subinterval  $[x_{i-1}, x_i]$ . It can be shown that the area of such a trapezoid with one edge as a quadratic curve is  $(f(x_{i-1}) + 4f(\bar{x}_i) + f(x_i))h/6$ . Thus,

$$\begin{aligned} \int_a^b f(x)dx &= \sum_{i=1}^n \int_{x_{i-1}}^{x_i} f(x)dx \\ &\approx \sum_{i=1}^n \frac{f(x_{i-1}) + 4f(\bar{x}_i) + f(x_i)}{6} h \\ &= \frac{1}{3} \sum_{i=1}^n \frac{f(x_{i-1}) + f(x_i)}{2} h + \frac{2}{3} \sum_{i=1}^n f(\bar{x}_i)h. \end{aligned}$$

Notice that the first summation in the formula is the same as the Trapezoidal Rule. Now it can be defined as

```
double simpson(double a, double b, pfn f, int n) {
    double h = (b - a)/n;
    double sum = f(a)*0.5;
    for (int i = 1; i < n; i++) sum += f(a + i*h);
    sum += f(b)*0.5;

    double summid = 0.0;
    for (int i = 1; i <= n; i++) summid += f(a + (i-0.5)*h);

    return (sum + 2*summid)*h/3.0;
}
```

In §5.1, an abstract mechanism called *class* is used to present this example in a different style. Notice that this C-style definition of *trapezoidal()* and *simpson()* passes a function pointer *f* for the integrand as an argument, which imposes function calling overhead for each call *f()* inside *trapezoidal()* and *simpson()*. In §7.7 some efficient techniques are applied to overcome the function calling overhead in passing a function pointer to another function call, where integration functions are written with a type parameter so that they can also handle arithmetics in different precisions such as *float*, *double*, and *long double*. For more details on mathematical aspects of the Trapezoidal and Simpson's Rules, see [Ste99, KC96, VPR00].

## 3.14 Exercises

- 3.14.1. Write a program that defines a *const* integer *n*, generates *n* random numbers and stores them in an array, and computes and outputs the average of these random numbers. Also compute and output the maximum and minimum values of the random numbers generated.
- 3.14.2. The standard library *<assert.h>* contains a macro *assert()* that can be used to ensure that the value of an expression is as expected. Here is an example of how *assert()* can be used to ensure a variable *n* is positive, but less than or equal to 100:

```
#include <assert.h>
void f(int n) {
    assert(n > 0 && n <= 100);
}
```

If an assertion fails, the system will print out a message and abort the program. When the macro *NDEBUG* is defined at the point where *<assert.h>* is included, for example,

```
#define NDEBUG
#include <assert.h>
```

then all assertions are ignored. Thus assertions can be freely used during program development and later discarded (for run-time efficiency) by defining *NDEBUG*. Write a program to generate Fibonacci numbers (see §2.4) ensuring that they are positive using an assertion.

- 3.14.3. Write a function having prototype

```
int sumsq(int n, int m);
```

that returns the sum of squares  $n^2 + (n+1)^2 + \cdots + (m-1)^2 + m^2$  for  $n < m$ . The function should also be able to handle the case  $n > m$ .

- 3.14.4. Write a function that computes and returns the number of (decimal) digits of a given integer  $n$ . For example, it should return 3 if  $n = 145$ .
- 3.14.5. Write a function that outputs (e.g., the first 1000 terms of) the sequence  $d_n = \sqrt{n} - \sqrt{n-1}$  for  $n = 1, 2, 3, \dots$ . Write another function that computes  $d_n$  in a mathematically equivalent but numerically more robust way:

$$d_n = \frac{1}{\sqrt{n} + \sqrt{n-1}} \quad \text{for } n = 1, 2, 3, \dots$$

Compare the difference in the outputs of the two functions. The second one should give more accurate results since it avoids subtracting two nearly equal numbers and thus reduces cancellation errors.

- 3.14.6. An algorithm to find a square root of a positive real number  $b$  is:

$$x_0 = 1, \quad x_{n+1} = \frac{1}{2} \left[ x_n + \frac{b}{x_n} \right], \quad n = 0, 1, 2, \dots$$

Write a function that implements the algorithm. A fast (quadratic) convergence should be observed for  $x_n \rightarrow \sqrt{b}$ . Compare the accuracy of this function with the built-in function `sqr()` in `<math.h>`.

- 3.14.7. Write a function that takes three *doubles*  $a$ ,  $b$ , and  $c$  as input and computes the roots of the quadratic equation

$$ax^2 + bx + c = 0.$$

When  $a = 0$  and  $b \neq 0$  the equation has only one root  $x = -c/b$ . When  $a \neq 0$ , its roots are given by

$$x_1 = (-b + \sqrt{b^2 - 4ac})/(2a), \quad x_2 = (-b - \sqrt{b^2 - 4ac})/(2a).$$

If the discriminant  $b^2 - 4ac$  is negative, the two roots are complex numbers. Output complex numbers in the form of  $r + mi$ , where  $i = \sqrt{-1}$  and  $r$  and  $m$  are real and imaginary parts. Write a `main()` program to test your function.

- 3.14.8. The statements

```
a[i++] *= n;
a[i++] = a[i++] * n;
```

are not equivalent. The operand `a[i++]` has a side effect of incrementing  $i$ . Except for possible implementation efficiency, the first statement is equivalent to

```
a[i] = a[i]*n;
i = i + 1;
```

while the second statement is equivalent to either

```
a[i+1] = a[i]*n;
i = i + 2;
```

or

```
a[i] = a[i+1]*n;
i = i + 2;
```

depending on whether the right-hand side of the assignment is evaluated before the left-hand side. The order of their evaluations is not defined. Test what result your machine gives for the second statement above. Such a statement is not portable and should be avoided.

- 3.14.9. Allocate space for a sequence, denoted by  $s$ , of  $2n + 1$  *doubles* so that the index changes from  $-n$  to  $n$ , where  $n$  is a given positive integer. Assign  $s[i] = i$  for  $i = -n, -(n-1), \dots, -1, 0, 1, \dots, n-1, n$ , and print out the values of  $s[i]$ ,  $-n \leq i \leq n$  for  $n = 10$ . Finally, delete the space.

- 3.14.10. Explain the difference between a two-dimensional array and a double pointer. Consider further an example of a two-dimensional array:

```
double tda[5][7];
```

The base address of the array  $tda$  is  $\&tda[0][0]$ , or  $tda[0]$ , but not  $tda$ . The array name  $tda$  by itself is equivalent to  $\&tda[0]$ . What can you say for a double pointer and a one-dimensional array?

- 3.14.11. Write a function that takes an  $n$  by  $m$  matrix and a vector of  $m$  components as input and computes the matrix-vector product such that it has prototype:

```
void multiply(const double** const mx,
             const double* const vr, int n,
             int m, double* const pt);
```

Here the double pointer  $mx$  represents the matrix, single pointer  $vr$  the input vector, and  $pt$  the matrix-vector product. Write a *main()* program to test the function. Space should have been allocated for  $mx$ ,  $vr$ , and  $pt$  before calling the function.

- 3.14.12. Declare an  $n$  by  $n$  upper triangular matrix *utm* of *floats* using a double pointer. To save memory, store only the upper part of the matrix. The entry at row  $i$  and column  $j$  should be accessed by using *utm*[ $i$ ][ $j$ ] for  $i = 0, 1, \dots, n-1$  and  $j = i, i+1, \dots, n-1$ . Then assign *utm*[ $i$ ][ $j$ ] =  $2i + j$  for  $n = 4$  and print out all the entries. Finally, deallocate space for *utm*. To save memory, space should not be allocated for the lower triangular part of the matrix.
- 3.14.13. Write a function that takes an  $n$  by  $n$  upper triangular matrix and a vector of  $n$  components as input and computes the matrix-vector product. It may have either of the prototypes

```
double* multiplyutm(const double** const utm,
                  const double* const vr, int n);
void multiplyutm(const double** const utm,
                const double* const vr,
                int n, double* const pt);
```

Implement the matrix as in Exercise 3.14.12 and the vector as a pointer. Write a *main()* program to test the function.

- 3.14.14. A three-dimensional array with dimensions  $p \times q \times r$  using a triple pointer can be allocated and deallocated as

```
int p = 4, q = 2, r = 3;          // or computed at run-time

double*** p3d = new double** [p]; // allocate space
for (int i = 0; i < p; i++) {
    p3d[i] = new double* [q];
    for (int j = 0; j < q; j++) p3d[i][j] = new double [r];
}

for (int i = 0; i < p; i++) // access its entries
    for (int j = 0; j < q; j++)
        for (int k = 0; k < r; k++) p3d[i][j][k] = i + j + k;

for (int i = 0; i < p; i++) { // delete space after use
    for (int j = 0; j < q; j++) delete[] p3d[i][j];
    delete[] p3d[i];
}
delete[] p3d;
```

Note that the order of memory deallocation should be the opposite of allocation. Allocate space for a four-dimensional array of dimensions  $p \times q \times r \times s$  using a quadruple pointer:



```
double**** p4d;
```

Assign it some values such as  $p4d[i][j][k][m] = i - j + k - m$ . Finally, deallocate the space.

- 3.14.15. Write a recursive function having the prototype

```
long fibonacci(int n);
```

that takes an integer  $n$  and calculates the Fibonacci number  $f_n$ ; see §2.4 for the definition of Fibonacci numbers. Also output the number of recursions needed to obtain  $f_n$ .

- 3.14.16. Write a function to print out a decimal integer  $v$  in base  $b$  having the prototype:

```
void print(int v, int b = 10); // b defaulted to be 10
```

with  $2 \leq b \leq 16$ . Your program should print out an error message when a user attempts to input  $b$  outside this range. Hint: an integer  $v$  can be expressed in base  $b$  as  $a_n a_{n-1} \cdots a_1 a_0$  if  $v = a_0 + a_1 b + a_2 b^2 + \cdots + a_n b^n$  with  $0 \leq a_i < b$ ,  $i = 0, 1, \dots, n$ , and  $a_n \neq 0$ .

- 3.14.17. Write the polynomial

$$p_8(x) = x^8 - 8x^7 + 28x^6 - 56x^5 + 70x^4 - 56x^3 + 28x^2 - 8x + 1$$

directly into a nested multiplication form using the idea of Horner's algorithm. Then print out values of  $p_8(x)$  for  $x$  taking on 11 equally spaced values covering the interval  $[0.99999, 1.0001]$ . You should get the same values as the function *horner()*, defined in §3.12.

- 3.14.18. A man has just started a fixed-rate mortgage of total amount of  $S$  dollars with annual percentage rate  $r$  and length of  $m$  months. But he decides to pay off the mortgage early and pay  $P_1, P_2, \dots$ , and  $P_{n-1}$  dollars for the first, second,  $\dots$ , and  $(n-1)$ th month, respectively, and pay off the mortgage at the end of the  $n$ th month with  $n \leq m$ . The payoff amount  $P_n$  can be computed from the formula:

$$S = \sum_{i=1}^{n-1} P_i \left(1 + \frac{r}{12}\right)^{-i} + P_n \left(1 + \frac{r}{12}\right)^{-n},$$

assuming that each  $P_i$  is no smaller than his regular monthly payment so that no penalty is imposed on him. Write a function that computes the payoff amount  $P_n$ , given  $S, r, P_1, P_2, \dots$ , and  $P_{n-1}$ . If the man has a 15-year mortgage of 200000 dollars at rate 0.0625, but pays  $P_i = 2000 + 10 * i$  dollars in month  $i$  for  $i = 1, 2, \dots, 59$ , and wants to pay off the mortgage in month 60, what is his payoff amount  $P_{60}$ ? Hint: Horner's algorithm should be used in calculating the summation.

- 3.14.19. Apply the idea of Horner's algorithm to derive efficient algorithms for evaluating

$$B_n = \sum_{i=0}^n \prod_{j=0}^i b_j = b_0 + b_0 b_1 + b_0 b_1 b_2 + \cdots + b_0 b_1 \cdots b_n,$$

$$D_n = \sum_{i=0}^n \prod_{j=0}^i \frac{1}{d_j} = \frac{1}{d_0} + \frac{1}{d_0 d_1} + \frac{1}{d_0 d_1 d_2} + \cdots + \frac{1}{d_0 d_1 \cdots d_n}.$$

The algorithms should require only  $n$  multiplications for  $B_n$  and  $n+1$  divisions for  $D_n$ , in addition to  $n$  additions for each. Then write functions having prototypes

```
double sumntp(const double* const b, int n);
double sumdiv(const double* const d, int n);
```

for them, where  $b$  is a pointer that points to the numbers  $b_0, b_1, \dots, b_{n-1}, b_n$  and  $d$  points to  $d_0, d_1, \dots, d_{n-1}, d_n$ . Use both functions to compute the partial sums:

$$E_m = \sum_{n=2}^m \frac{1}{n!}$$

for  $m = 100, 500$ , and  $1000$ . Hint: your answer should be closer to 0.71828182845904523536 as  $m$  gets larger.

- 3.14.20. The base of the natural algorithm  $e$  is

$$e = 2.7182818284590452353602874713526624977572 \dots$$

to 41 significant digits. It is known that

$$x_n = \left[1 + \frac{1}{n}\right]^n \rightarrow e \quad \text{as } n \rightarrow \infty.$$

Write a function that generates the sequence  $\{x_n\}$  and prints out the error  $e - x_n$ . A very slow (worse than linear) convergence should be observed for  $x_n \rightarrow e$ . Indeed it can be shown that  $|e - x_{n+1}|/|e - x_n| \rightarrow 1$  as  $n \rightarrow \infty$ . A computationally effective way to compute  $e$  to arbitrary precision is to use the infinite series:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = 2 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots.$$

To reduce roundoff errors, smaller terms should be added first in calculating partial sums. It can be shown that

$$\left| e - \sum_{n=0}^m \frac{1}{n!} \right| < \frac{e}{(m+1)!}.$$

Write another function to generate the partial sums for approximating  $e$  and compare the convergence. Horner's algorithm may be used to efficiently compute this kind of partial sum.

- 3.14.21. When  $n$  is large, the factorial  $n!$  can easily cause *overflow*. Write a function to calculate  $n!$  by storing a large integer in an array of integers, whose one element stores one digit of the large integer. To illustrate the idea, consider the program:

```
#include <iostream>      // for input and output
#include <stdlib.h>      // for function exit()
const int mxdgs = 9500; // max # of digits in factorial
int main() {
    int n;
    cout << "Enter a positive integer: \n";
    cin >> n;

    // digits of n! will be stored in array dgs backwards,
    // ie, highest digit will be stored in dgs[0], next
    // highest in dgs[1], and so on.
    int dgs[mxdgs];
    dgs[0] = 1;
    for (int i = 1; i < mxdgs; i++) dgs[i] = 0;
    for (int k = 2; k <= n; k++) { // n! = 2*3*4... (n-1)*n
        int cary = 0;
        for (int j = 0; j < mxdgs; j++) {
            // multiply k with every digit of (k-1)! to get k!
            dgs[j] = k*dgs[j] + cary;
            cary = dgs[j]/10;
            dgs[j] -= 10*cary;
        }
    }

    // counting the number of digits in n!
    int count = mxdgs - 1;
    while (!dgs[count]) count--;

    // compute the number of digits in integer n
    int numdgofn = 0;
    int m = n;
    while (m) {
        m /= 10;
        numdgofn++;
    }
}
```

```

// check to see if size of dgs is enough
if (mxdgs - count < numdgofn + 1) {
    cout << "factorial overflow, adjust parameter.\n";
    exit(1);          // exit program
}

// printing n!: lowest digit dgs[count] first,
// highest digit dgs[0] last.
cout << "There are " << count + 1 << " digits in "
    << n << "! . They are:\n";
for (int i = count; i >= 0; i--) cout << dgs[i];
cout << '\n';
}

```

The standard library function *exit()*, declared in *<stdlib.h>*, is called (and returns value 1) to terminate the program when the *dgs* array is not large enough to store  $n!$ . In this program, the factorial  $(k-1)!$  is stored digit by digit in an array of integers. To get  $k!$ , multiply every element of the array (every digit of  $(k-1)!$ ) by  $k$ . The result is still stored in the array. Looping  $k$  from 2 through  $n$  gives  $n!$ . Notice that the digits of  $k!$  are stored in the array backwards. For example, the three digits 120 of  $5!$  are stored backwards as 021000...000 with  $dgs[0] = 0, dgs[1] = 2, dgs[2] = 1, dgs[3] = 0, \dots, dgs[mxdgs-1] = 0$ . With this technique,  $3000!$ , which has 9131 digits, can be calculated correctly on a computer with *sizeof(int)* = 4. To compute the factorial of a larger (than 3000) integer, the size of the array can be adjusted.

Write a factorial function having prototype:

```
facts factorial(int n, int mxdgs);
```

where *facts* is a structure

```

struct {
    int* digits;          // store the digits of factorial
    int numdgs;          // number of digits of factorial
};

```

This function returns a *struct* that contains the number of digits of  $n!$  and the digits of  $n!$ . It should print out an error message and exit when  $n!$  has more than *mxdgs* digits.

- 3.14.22. Apply the Trapezoidal and Simpson's Rules to evaluate the integral  $\int_0^2 e^{-x^2} dx$ , where  $e$  is the base of natural logarithm. Output the approximate values corresponding to the number of subintervals

$n = 100, 300$ , and  $500$ . Compare your answers with  $0.8820813907$  (obtained from a handbook).

- 3.14.23. Compute the integral  $\int_1^2 ((\sin x)/x)dx$  with five digit accuracy after the decimal point.
- 3.14.24. The standard library `<stdlib.h>` contains a sorting function `qsort()`, based on the quick sort algorithm. It can sort an array of any elements in an order defined by the user through a pointer to a function. It has prototype:

```
typedef int (*CMF)(const void* p, const void* q);
// a type for comparing p and q, it defines an order
void qsort(void* b, size_t n, size_t sz, CMF cmp);
// sort "n" elements of array "b" in an order
// defined by comparison function "cmp".
// Each element of "b" is of size "sz" (in bytes).
```

Here `size_t` is an unsigned integral type (possibly *unsigned long*) defined in `<stddef.h>` and used in the standard libraries.

If an array of `point2d` (see §3.4) is going to be sorted in increasing order according to its  $x$ -coordinate, a comparison function can be defined as

```
int cmp2dx(const void* r, const void* s) {
    double d = static_cast<const point2d*>(r)->x -
               static_cast<const point2d*>(s)->x;
    if (d > 0) return 1;
    else if (d < 0) return - 1;
    else return 0;
}
```

The operator `static_cast` converts between related types such as from one pointer type to another or from an enumeration to an integral type. Since `r` has type `const void*`, it needs to be converted to `const point2d*` before accessing its member `x`.

Now arrays of `point2d` can be generated and sorted by `qsort()` in the order defined by `cmp2dx()` :

```
int main() {
    const int n = 10;
    point2d a[n];
    for (int i = 0; i < n; i++) {
        a[i].x = i*(5 - i); a[i].y = 5/(i+2.3) - 1;
    }
}
```

```

    qsort(a, n, sizeof(point2d), cmp2dx);
    for (int i = 0; i < n; i++)
        cout << a[i].x << " " << a[i].y << '\n';
}

```

For people who are curious about how such a sorting function is defined, here is a definition of a related (based on the shell sort algorithm; see [Knu98] and [Str97]) function:

```

void shsort(void* b, size_t n, size_t sz, CMF cmp) {
    for (int gap = n/2; gap > 0; gap /= 2)
        for (int i = gap; i < n; i++)
            for (int j = i - gap; j >= 0; j -= gap) {
                char* base = static_cast<char*>(b);
                char* pj = base + j*sz;
                char* pjpg = base + (j + gap)*sz;
                if (cmp(pj, pjpg) > 0) { // swap b[j], b[j+gap]
                    for (int k = 0; k < sz; k++) {
                        char temp = pj[k];
                        pj[k] = pjpg[k];
                        pjpg[k] = temp;
                    }
                }
            }
}

```

Write a program that sorts an array of *point2d* in decreasing order according to its *y*-coordinate by using the standard library function *qsort()* or the user-defined function *shsort()* above. Note that C++ provides a more efficient mechanism for dealing with objects of a generic type; see Chapters 7 and 10.

3.14.25. Generate an array of random numbers and mimic Exercise 3.14.24 to sort this array in decreasing order in terms of absolute value.

3.14.26. Consider the declaration:

```
int* (*(a)[10])(double);
```

What is the type of *a*? It can be analyzed by substitution as follows. Let *b* = *\*(a)[10]*. Then the declaration can be written as

```
int* (b)(double);
```

Then  $b$  is a function that takes a *double* and returns a pointer to an *int*. Since  $b = *(*a)[10]$ , this relation can be written into a declaration:

```
b  *(*a)[10];
```

where  $b$  is a type just clarified above. It is equivalent to

```
b*  (*a)[10];
```

Thus  $a$  is a pointer to an array of 10 elements, each of which is a pointer to type  $b$ . Finally it can be concluded that  $a$  is a pointer to an array of 10 elements, each of which is a pointer to a function that takes a *double* and returns a pointer to an *int*. This substitution method can be rigorously justified in C++ by using the keyword *typedef* (why?).

Apply this idea to explain the type of  $x$  in the declaration:

```
double (*(*x())[10])(int);
```

using substitutions  $y = (*x())[10]$  and  $z = *x()$ . Hint:  $x$  is a function that takes no arguments and returns a pointer to type  $z$ , where  $z$  is an array of 10 elements of type  $y$ , and  $y$  is a pointer to a function that takes an *int* and returns a *double*. Such confusing declarations should be avoided in practice but are presented here for a better understanding of pointers, arrays, and functions.





# 4

## Namespaces and Files

C++ supports modular programming through a mechanism called *namespace*. A *namespace* is a logical unit that contains related declarations and definitions. The idea of modular programming is to divide a large program into small and logically related parts for easy management and information hiding. Dividing a large program into different parts and storing them in different files can also help to achieve modular programming (this is more so in C and FORTRAN 77). A few useful tools (some in UNIX and Linux) are also presented for managing files, creating a library, profiling a program, debugging a program, and timing a program. Towards the end of the chapter, two standard libraries on character strings and input and output streams are given. Finally, iterative algorithms for solving nonlinear equations are described.

### 4.1 Namespaces

The keyword *namespace* is used to group related declarations and definitions in C++. For example, we can group vector-related declarations in a namespace called *Vec* and matrix-related declarations in a namespace called *Mat*:

```

namespace Vec {
  const int maxsize = 100000;
  double onenorm(const double* const, int);    // 1-norm
  double twonorm(const double* const, int);    // 2-norm

```

```

    double maxnorm(const double* const, int);    // max norm
}

namespace Mat {
    double onenorm(const double** const, int);  // 1-norm
    double twonorm(const double** const, int);  // 2-norm
    double maxnorm(const double** const, int);  // max norm
    double frobnorm(const double** const, int); // Frobenius
}

```

For a vector  $v = [v_0, v_1, \dots, v_{n-1}]$ , its one, two, and maximum (also denoted by  $\infty$ ) norms are defined as

$$\begin{aligned}
 \|v\|_1 &= |v_0| + |v_1| + \dots + |v_{n-1}| = \sum_{i=0}^{n-1} |v_i|, \\
 \|v\|_2 &= (|v_0|^2 + |v_1|^2 + \dots + |v_{n-1}|^2)^{1/2} = \sqrt{\sum_{i=0}^{n-1} |v_i|^2}, \\
 \|v\|_\infty &= \max\{|v_0|, |v_1|, \dots, |v_{n-1}|\} = \max_{0 \leq i < n} |v_i|,
 \end{aligned}$$

respectively. For example, for the vector  $u_4 = [1, -2, 0, 10]$ ,

$$\begin{aligned}
 \|u_4\|_1 &= |1| + |-2| + |0| + |10| = 13, \\
 \|u_4\|_2 &= (|1|^2 + |-2|^2 + |0|^2 + |10|^2)^{1/2} = \sqrt{105}, \\
 \|u_4\|_\infty &= \max\{|1|, |-2|, |0|, |10|\} = 10.
 \end{aligned}$$

For an  $n$  by  $n$  matrix  $A = (a_{i,j})_{i,j=0}^{n-1}$ , its one,  $\infty$ , and Frobenius norms are defined as

$$\begin{aligned}
 \|A\|_1 &= \max \left\{ \sum_{i=0}^{n-1} |a_{i,0}|, \sum_{i=0}^{n-1} |a_{i,1}|, \dots, \sum_{i=0}^{n-1} |a_{i,n-1}| \right\} = \max_{0 \leq j < n} \sum_{i=0}^{n-1} |a_{i,j}|, \\
 \|A\|_\infty &= \max \left\{ \sum_{j=0}^{n-1} |a_{0,j}|, \sum_{j=0}^{n-1} |a_{1,j}|, \dots, \sum_{j=0}^{n-1} |a_{n-1,j}| \right\} = \max_{0 \leq i < n} \sum_{j=0}^{n-1} |a_{i,j}|, \\
 \|A\|_F &= (|a_{0,0}|^2 + |a_{0,1}|^2 + \dots + |a_{n-1,n-1}|^2)^{1/2} = \sqrt{\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} |a_{i,j}|^2},
 \end{aligned}$$

respectively. For example, for the matrix

$$B_3 = \begin{bmatrix} 1 & -2 & 0 \\ -4 & 3 & 5 \\ 2 & 0 & 6 \end{bmatrix},$$

its one, max, and Frobenius norms are:

$$\begin{aligned} \|B_3\|_1 &= \max\{|1| + |-4| + |2|, |-2| + |3| + |0|, |0| + |5| + |6|\} \\ &= \max\{7, 5, 11\} = 11, \\ \|B_3\|_\infty &= \max\{|1| + |-2| + |0|, |-4| + |3| + |5|, |2| + |0| + |6|\} \\ &= \max\{3, 12, 8\} = 12, \\ \|B_3\|_F &= \sqrt{|1|^2 + |-2|^2 + |0|^2 + |-4|^2 + |3|^2 + |5|^2 + |2|^2 + |0|^2 + |6|^2} \\ &= \sqrt{95}. \end{aligned}$$

That is, the 1-norm of a matrix is the maximum of the absolute column sums,  $\infty$ -norm is the maximum of the absolute row sums, and the Frobenius norm is the square root of the square sum of all entries. The 2-norm of a matrix is more involved and not mentioned any further. These norms provide certain measurements of the magnitude of a vector or matrix.

A namespace is a scope. Members of a namespace can be accessed by using the `::` operator. In particular, the functions of namespace *Vec* can be defined this way.

```
double Vec::onenorm(const double* const v, int size) {
    if (size > maxsize) cout << "vector size too large.\n";
    double norm = fabs(v[0]);          // fabs() in <math.h>
    for (int i = 1; i < size; i++) norm += fabs(v[i]);
    return norm;
}

double Vec::maxnorm(const double* const v, int size) {
    if (size > maxsize) cout << "vector size too large.\n";
    double norm = fabs(v[0]);
    for (int i=1; i<size; i++) norm = max(norm, fabs(v[i]));
    return norm;                       // max() in <algorithm>
}

double Vec::twonorm(const double* const v, int size) {
    if (size > maxsize) cout << "vector size too large.\n";
    double norm = v[0]*v[0];
    for (int i = 1; i < size; i++) norm += v[i]*v[i];
    return sqrt(norm);
}
```

The function *fabs()*, declared in *<math.h>*, gives the absolute value of its argument and *max()*, declared in *<algorithm>*, returns the maximum of its two arguments. The name *maxsize* can be used directly without using the qualification operator `::` inside the definitions of members of namespace *Vec*. This is because *maxsize* is a member of *Vec*. Note that in the third function definition above, the variable *norm* could cause overflow while *sqrt(norm)*

is still within the range of the machine. A robust 2-norm formula is given in Exercise 4.8.5 that will not cause overflow if  $\|v\|_2$  is within range.

Functions for the matrix namespace *Mat* can be defined similarly. However, when members from another namespace are needed, the qualification operator `::` should be used. For example, use *Vec::onenorm* to refer to the function *onenorm()* in the namespace *Vec*. In this way, name clashes can be avoided. That is, members of different namespaces can have the same names. For example,

```
double Mat::maxnorm(const double** const a, int size) {
    double norm = Vec::onenorm(a[0],size);    // 1-norm of row 0
    for (int i = 1; i < size; i++)
        norm = max(norm, Vec::onenorm(a[i],size)); //a[i]: row i
    return norm;
}

double Mat::onenorm(const double** const a, int size) {
    double norm = 0;
    for (int j = 0; j < size; j++) {
        double temp = 0;                // store column abs sum
        for (int i = 0; i < size; i++) temp += fabs(a[i][j]);
        norm = max(norm, temp);
    }
    return norm;
}

double Mat::frobnorm(const double** const a, int size) {
    double norm = 0;
    for (int i = 0; i < size; i++)
        for (int j = 0; j < size; j++) norm += a[i][j]*a[i][j];
    return sqrt(norm);
}
```

See Exercise 4.8.6 for a robust definition of the Frobenius matrix norm. However, for matrices whose entries are not large enough to cause overflow when summing the squares of all entries, this straightforward evaluation of the Frobenius norm should be more efficient.

Now the namespaces *Vec* and *Mat* can be applied to evaluate norms of matrices and vectors:

```
// declarations and definitions of Vec and Mat may be put here

int main() {
    int n = 1000;
    double** a = new double* [n];
    for (int i = 0; i < n; i++) a[i] = new double [n];
```

```

for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++) a[i][j] = 1.0/(i + j + 1);

cout << Mat::maxnorm(a, n) << '\n';    // matrix max norm
cout << Vec::maxnorm(a[2], n) << '\n';  // vector max norm
}

```

Notice that  $a[2]$  refers to row 2 of matrix  $a$ , which can be treated as a vector. A matrix  $H_n = (h_{i,j})_{i,j=0}^{n-1}$  with entries  $h_{i,j} = 1/(i + j + 1)$  is called a *Hilbert* matrix with dimension  $n$ .

The declarations in a namespace are called the interface part and the definitions of the members are called the implementation part. A normal user need only see the interface and figure out how to use it. The implementation part is usually placed somewhere else. For large programs, it has the advantage that the user will not be confused by the implementation details and can just concentrate on the interface. The larger a program is, the more useful namespaces are to express logical separation of its parts.

Since a namespace is a scope, the usual scope rules hold for it. If a name is previously declared in a namespace, it can be used directly inside the namespace and its members. For example, the member *maxsize* of *Vec* can be used directly in the definition of *Vec::onenorm()*. A name from another namespace can be used when qualified by the name of its namespace. For example, the name of namespace *Vec* must be used to qualify its function *onenorm()* in order for it to be recognized in namespace *Mat*.

#### 4.1.1 Using Declarations and Directives 指令

When namespace *Mat* or other programs have to use members of namespace *Vec* intensively, repeated qualification using *Vec::* can be tedious and distracting, especially if a namespace has a long name. This qualification can be eliminated by applying the keyword *using*:

```

double Mat::maxnorm(const double** const a, int size) {
    using Vec::onenorm;    // use onenorm from namespace Vec

    double norm = onenorm(a[0],size); // Vec::onenorm is used
    for (int i = 1; i < size; i++)
        norm = max(norm, onenorm(a[i],size));
    return norm;
}

```

The *using* declaration brings *Vec::onenorm* into the current scope and then it can be used directly without qualifying by the name of its namespace. Note that a function name overloading also happened here. Since *Vec*'s function *onenorm()* takes different arguments from *Mat::onenorm()*, the compiler can figure out that *Vec::onenorm()* is called. If both *Vec* and *Mat*

had exactly the same members, the *using*-declaration would cause a name clash.

If a member of *Mat* depends on several members of *Vec*, then the *using*-declaration has to be used for each member. This can be replaced by the *using*-directive:

```
namespace Mat {
    double onenorm(const double** const, int);    // 1-norm
    double twonorm(const double** const, int);    // 2-norm
    double maxnorm(const double** const, int);    // max norm
    double frobnorm(const double** const, int);    // Frobenius

    using namespace Vec;    // make all names from Vec available
}
```

The *using* directive brings all members of namespace *Vec* into the scope of namespace *Mat*. Then all names of *Vec* can be used directly in namespace *Mat* and the definitions of its members.

By the *using*-directive for namespace *Vec*, the code for members of *Mat* can then be simplified as if everything of *Vec* had the same scope. For example, *Mat::maxnorm()* can be defined as:

```
double Mat::maxnorm(const double** const a, int size) {
    double norm = onenorm(a[0],size);    // Vec::onenorm is used
    for (int i = 1; i < size; i++)
        norm = max(norm, onenorm(a[i],size));
    return norm;
}
```

If two namespaces have two functions that have the same prototype and both namespaces are used in a third namespace, it is possible to choose one function over the other and to avoid name clash. For example,

```
namespace A {
    int f(int);
}

namespace B {
    int f(int);
}

namespace C {
    using namespace A;
    using namespace B;
    using A::f;    // use A's member f() instead of B::f()

    int g(int i) {
```

```

    return i + f(i);          // A::f(i) is called
}

int h(int i) {
    return i + B::f(i);      // B::f(i) is called
}
}

```

In the definition of  $C::g()$ , the function  $A::f()$  is used because of the *using* declaration of  $A::f$ . Here the functions  $g()$  and  $h()$  are defined inside the namespace  $C$ . Generally speaking, this is not preferred since it makes the interface of namespace  $C$  obscure, especially when  $C$  has a lot of members.

Namespaces enable codes written by different people to be combined easily. For example, two programmers might have used the same name for their variables with different meanings. Codes written by different people can be grouped into different namespaces and name clashes can be avoided. The *using*-declaration and *using*-directive can be used to easily access names in different namespaces when names do not clash.

#### 4.1.2 Multiple Interfaces

If the namespace *Mat* needs to be expanded to include new function, data, or type members, we can combine old and new members to form a new version:

```

namespace Mat {
    double onenorm(const double** const, int); // 1-norm
    double twonorm(const double** const, int); // 2-norm
    double maxnorm(const double** const, int); // max norm
    double frobnorm(const double** const, int); // Frobenius

    double* eigenvalue(const double** const, int);
                                // find its eigenvalues
    double** inverse(const double** const, int);
                                // find inverse matrix
    void gausselim(const double** const, double* const, int);
                                // Gauss elimination
}

```

These two versions of the namespace *Mat* can coexist in a program so that each can be used where it is most appropriate. For example, for a user who just needs to evaluate matrix norms, the first version is sufficient to bring the functions of *Mat* into her scope (including what she does not need may be confusing). Minimal interfaces lead to systems that are easier to understand, have better information hiding, are easier to modify, and compile faster.

The second version is equivalent to:

```
namespace Mat {
    double* eigenvalue(const double** const, int);
                                // find its eigenvalues
    double** inverse(const double** const, int);
                                // find inverse matrix
    void gausselim(const double** const, double* const, int);
                                // Gauss elimination
}
```

provided the first version of *Mat* has been declared. That is, a namespace can have multiple interfaces, but the compiler will combine all of them together for the namespace. Another example is:

```
namespace A {
    struct pt { double x; double y; };
    int f(pt);    // A has two members pt and f()
}

namespace A {
    void g();    // now A has three members pt, f(), and g()
}
```

The second version of namespace *A* is equivalent to:

```
namespace A {
    struct pt { double x; double y; };
    int f(pt);
    void g();
}
```

#### 4.1.3 Namespace Alias

Aliases for long namespace names can be created. For example,

```
namespace Numerical_Integration { // name is very long
    double lowerbound;           // lower integral bound
    double (*integrand)(double); // integrand function
    // .....
}
```

```
Numerical_Integration::lowerbound = 1.0;
```

The long name of this namespace takes time to type and is not quite readable. A short name *NI* can be provided for the namespace:

```
namespace NI = Numerical_Integration;
```



```

NI::lowerbound = 1.0;

namespace my_module {
    using NI::lowerbound;
    // .....
}

```

The short name *NI* is very convenient to use.

#### 4.1.4 Unnamed Namespaces

Sometimes we do not want the name of a namespace to be known outside a local context. An unnamed namespace, which is a namespace without a name, can be used. For example,

```

namespace {
    int a;
    void f(int& i) { i++; }
    int g() { return 5; }
}

```

Its members are global names in the file (more precisely, *translation unit*; see §4.3) where it is defined. It is equivalent to

```

namespace anyname {
    int a;
    void f(int& i) { i++; }
    int g() { return 5; }
}
using namespace anyname;

```

where *anyname* is some name unique to the scope in which the namespace is defined. Unnamed namespaces in different translation units (files; see §4.3) are distinct. There is no way to refer to a member of an unnamed namespace from another translation unit. Unnamed namespaces can be used to avoid using one of the meanings of *static* in C programs (see §4.3.2).

#### 4.1.5 Name Lookup

If a function can not be found in the scope where it is used, looking up of the namespaces of its arguments will follow. For example,

```

namespace Matrix {
    struct mat {
        double** elem;
        int size;
    };
    double maxnorm(const mat& a) { /* it may be defined here */ }
}

```

```

    double onenorm(const mat& a);
}

double f(Matrix::mat a) {
    double m = maxnorm(a);      // Matrix::maxnorm() is called
    double n = onenorm(a);      // Matrix::onenorm() is called
    return m+n;
}

```

In the call *maxnorm(a)*, the function *maxnorm()* is not defined in the scope. Then the compiler looks for it in its argument *a*'s namespace *Matrix*. It can also be explicitly qualified by the name of the namespace as *Matrix::maxnorm(a)*. This lookup rule not only saves a lot of typing compared with explicit qualification, but also makes the code look more concise.

## 4.2 Include Files

This section deals with various issues on how to include standard library header files and user-defined header files.

### 4.2.1 Include Files for Standard Libraries

The way in which a C program includes its standard libraries' header files is still valid in C++. For example,

```

#include <math.h>           // math library, inherited from C
#include <stdio.h>          // I/O library, from C
int main() {
    printf("Hello World!\n");    // prints string to screen
    printf("square root of pi is: %f", sqrt(atan(1)*4));
}

```

where the output function *printf()* has the prototype (it is an example of a function that takes an arbitrary number of arguments, indicated by an ellipsis):

```
int printf(const char* ...);           // from <stdio.h>
```

It prints out strings and values of variables to the screen according to some specified format. Here *%f* means to print out the value of *sqrt(atan(1)\*4)* as a *float* and *%* is a format control character.

All C++ standard libraries, including those inherited from C, are in a namespace called *std*. However, there is no namespace in C. The declarations of standard C I/O facilities from the C header file *<stdio.h>* are wrapped up in the namespace *std* like this:

```
// in header file stdio.h
namespace std {

    int printf(const char* ...);
    // ..... declarations of C's other stuff in stdio.h
}
```

```
using namespace std;
```

The *using*-directive above makes every declaration in *stdio.h* globally available in every file that includes the header *stdio.h*.

Similarly, the declarations in C's *<math.h>* file are put in the namespace *std*, and made globally available. New C++ header files such as *<cstdio>* and *<cmath>* (note: without the .h suffix) are defined for users who do not want the names globally available. For example, the C++ header file *cstdio* looks like

```
// in header file cstdio
namespace std {
    int printf(const char* ...);
    // ..... declarations of C's other stuff in cstdio
}
```

Without the *using*-directive as opposed to *<stdio.h>*, all the declarations in *<cstdio>* are restricted to the namespace *std*.

In general, for each C standard library *<X.h>*, there is a corresponding standard C++ header *<cX>*, inherited from C. If we use C++'s headers, our C code above should go this way:

```
#include <cmath>          // C++ math library
#include <cstdio>          // C++ I/O (input/output) library
int main() {
    std::printf("Hello World!\n"); // print string to screen
    std::printf("square root of pi is: %f",
                std::sqrt(std::atan(1)*4));
}
```

That is, explicit namespace qualification such as *std :: printf()* and *std :: sqrt()* must be used here. If we do not want the namespace qualifier *std ::* we can either use C-style include files with a suffix *.h* or do this:

```
#include <cmath>          // C++ math library
#include <cstdio>          // C++ I/O (input/output) library
int main() {
    using namespace std; // make declarations in std available
    printf("Hello World!\n"); // print string to screen
    printf("square root of pi is: %f", sqrt(atan(1)*4));
}
```

<code>&lt;assert.h&gt;</code>	diagnostics, defines <code>assert()</code> macro
<code>&lt;ctype.h&gt;</code>	character handling
<code>&lt;errno.h&gt;</code>	errors and error handling
<code>&lt;float.h&gt;</code>	floating point limits
<code>&lt;limits.h&gt;</code>	integer limits
<code>&lt;locale.h&gt;</code>	localization
<code>&lt;math.h&gt;</code>	mathematics
<code>&lt;setjmp.h&gt;</code>	nonlocal jumps
<code>&lt;signal.h&gt;</code>	signal handling
<code>&lt;stdarg.h&gt;</code>	variable arguments
<code>&lt;stddef.h&gt;</code>	common definitions
<code>&lt;stdio.h&gt;</code>	standard input and output
<code>&lt;stdlib.h&gt;</code>	general utilities
<code>&lt;string.h&gt;</code>	string handling
<code>&lt;time.h&gt;</code>	date and time

TABLE 4.1. Standard C header files.

Note that C++ has an alternative input and output library with headers `<iostream>` and `<fstream>`. Details of the standard libraries `<iostream>` and `<fstream>` are talked about in §4.6.

Table 4.1 lists standard C header files, while Tables 4.2 and 4.3 contain standard C++ header files.

#### 4.2.2 User's Own Include Files

By convention, standard libraries are included using angle brackets, and user's own header files are included using double quotation marks. For example, if a user's own header file *my.h* is like this:

```
// in file my.h

namespace MatVec {
    // .....          // declarations of matrix vector library
}
```

```
struct point2d {
    double x, y;
};          // note semicolon is needed after struct
```

then this header file should be included in another file using double quotes:

```
// in file my.cc

#include "my.h"          // user's own header file
```

<code>&lt;algorithm&gt;</code>	general algorithms
<code>&lt;bitset&gt;</code>	set of Booleans
<code>&lt;cassert&gt;</code>	diagnostics, defines <code>assert()</code> macro
<code>&lt;cctype&gt;</code>	C-style character handling
<code>&lt;cerrno&gt;</code>	C-style error handling
<code>&lt;cfloat&gt;</code>	C-style floating point limits
<code>&lt;climits&gt;</code>	C-style integer limits
<code>&lt;locale&gt;</code>	C-style localization
<code>&lt;cmath&gt;</code>	mathematical functions (real numbers)
<code>&lt;complex&gt;</code>	complex numbers and functions
<code>&lt;csetjmp&gt;</code>	nonlocal jumps
<code>&lt;csignal&gt;</code>	C-style signal handling
<code>&lt;cstdarg&gt;</code>	variable arguments
<code>&lt;cstdint&gt;</code>	common definitions
<code>&lt;cstdio&gt;</code>	C-style standard input and output
<code>&lt;cstdlib&gt;</code>	general utilities
<code>&lt;cstring&gt;</code>	C-style string handling
<code>&lt;ctime&gt;</code>	C-style date and time
<code>&lt;cwchar&gt;</code>	C-style wide-character string functions
<code>&lt;cwctype&gt;</code>	wide-character classification
<code>&lt;deque&gt;</code>	double-ended queue
<code>&lt;exception&gt;</code>	exception handling
<code>&lt;fstream&gt;</code>	streams to and from files
<code>&lt;functional&gt;</code>	function objects
<code>&lt;iomanip&gt;</code>	input and output stream manipulators
<code>&lt;ios&gt;</code>	standard <i>istream</i> bases
<code>&lt;iosfwd&gt;</code>	forward declarations of I/O facilities
<code>&lt;iostream&gt;</code>	standard <i>istream</i> objects and operations
<code>&lt;istream&gt;</code>	input stream
<code>&lt;iterator&gt;</code>	iterators and iterator support
<code>&lt;limits&gt;</code>	numeric limits, different from <code>&lt;climits&gt;</code>
<code>&lt;list&gt;</code>	doubly linked list
<code>&lt;locale&gt;</code>	represent cultural differences
<code>&lt;map&gt;</code>	associative array
<code>&lt;memory&gt;</code>	allocators for containers
<code>&lt;new&gt;</code>	dynamic memory management
<code>&lt;numeric&gt;</code>	numeric algorithms
<code>&lt;ostream&gt;</code>	output stream
<code>&lt;queue&gt;</code>	queue
<code>&lt;set&gt;</code>	set
<code>&lt;sstream&gt;</code>	streams to and from strings

TABLE 4.2. Standard C++ header files.

<code>&lt;stdexcept&gt;</code>	standard exceptions
<code>&lt;stack&gt;</code>	stack
<code>&lt;streambuf&gt;</code>	stream buffers
<code>&lt;string&gt;</code>	string, different from <code>&lt;cstring&gt;</code>
<code>&lt;typeinfo&gt;</code>	run-time type identification
<code>&lt;utility&gt;</code>	operators and pairs
<code>&lt;valarray&gt;</code>	numeric vectors and operations
<code>&lt;vector&gt;</code>	one-dimensional array

TABLE 4.3. Continuation of Standard C++ header files.

```

#include <cmath>           // C++ math library
#include <iostream>        // C++ I/O (input and output) library

int main() {
    point2d A;

    {
        using namespace std; // cmath, iostream available here
        A.x = sqrt(5.0);
        A.y = log(5.0);
        cout << "A.x = " << A.x << " " << "A.y = " << A.y;
    }
}

```

Spaces are significant within " " or < > of an include directive. For example,

```

#include " my.h"
#include <iostream >

```

will not find the header files *"my.h"* and *<iostream>*.

A simple way to partition a large program into several files is to put the definitions in a number of *.cc* files and to declare the types needed for them to communicate in a *.h* file that each *.cc* file includes. As a rule of thumb, a header file may contain items listed in Table 4.4, with an example in the second column. Conversely, a header file should not contain items listed in Table 4.5 (also with an example in the second column). Some of the items in these two tables are discussed later in this book.

### 4.2.3 Conditional Include Directives

The directive

named namespaces	namespace MatVec { /* ... */ }
type definitions	struct point2d { float x, y; };
data declarations	extern double b;
constant definitions	const double e = 2.718;
enumerations	enum color { red, yellow, green };
function declarations	extern int strlen(const char*);
inline function definitions	inline char get(char*p){return *p++;}
name declarations	struct Matrix;
include directives	#include <cmath>
macro definitions	#define VERSION 12
template declarations	template <class T> class A;
template definitions	template <class T> class B { /*...*/};
condition compile directives	#ifdef MatVec_H
comments	/* This is a matrix-vector library */

TABLE 4.4. Items that a header file may contain.

ordinary function defs	char get(char* p) { return *p++; }
data definitions	int aa; double bb = 4.5;
aggregate definitions	int table[] = { 1, 2, 3};
unnamed namespaces	namespace { int size = 100; /* ... */ }
exported template defs	export template <class T> f(T t){ /*...*/ }

TABLE 4.5. Items that a header file should not contain.

```
#ifndef identifier;
```

checks if *identifier* has been and remains defined. It can be used to include or skip some statements from it up to

```
#endif
```

depending on whether *identifier* has been and remains defined. For example, C and C++ can share the following header file.

```
#ifndef __cplusplus
namespace std {           // this line will be skipped
#endif                  // if __cplusplus is not defined

// declarations of standard library functions for C and C++
int printf(const char* ...);

#ifdef __cplusplus
}                          // skipped if __cplusplus not defined
using namespace std;      // skipped if __cplusplus not defined
#endif
```

The identifier `_plusplus` is defined when compiling (actually preprocessing) a C++ program and thus the function declarations such as `printf()` are put in the namespace `std`. When compiling a C program, however, `_plusplus` is not defined and there is no concept of *namespace*, the lines related to *namespace* will not be compiled.

Similarly, the conditional compilation directive

```
#ifndef identifier;
```

checks if *identifier* is still undefined.

A directive of the form

```
#if expression
```

checks whether the (constant) *expression* evaluates to *true*. Similar directives are `#else` and `#elif` (else if). They may be used as

```
#define DEBUG 1
```

```
// debugging information can be printed out.
```

```
#if DEBUG
```

```
    cout << "debug: value of a is: " << a << '\n';
```

```
#endif
```

After the debugging process, *DEBUG* can be defined to be 0 so that unnecessary information will not be printed out. In fact, this printing statement above will then be ignored by the preprocessor when the file is compiled. This use of directives actually serves as comments that can be easily turned on and off.

#### 4.2.4 File Inclusion

If a file includes a header file, the preprocessor simply copies the header file into the file at the point indicated before compilation begins. There are situations where several header files are necessary and one includes another. Then there is a danger that some header files are included in a file more than once. For example, if *head1.h* and *head2.h* both include *head.h* and *head1.h* also includes *head2.h*, to avoid chaining, *head.h* may take the form:

```
#ifndef _HEAD_H
#define _HEAD_H
```

#programa once

```
void my_function(double);
```

```
// declare/define other things that need be
```

```
// declared/defined in head.h
```

```
#endif
```



If the file *head.h* has already been included by the preprocessor, the macro `_HEAD_H` will have been defined and the body of the file will not be included more than once. For safety, every header file may be designed like this.

If somewhere the macro `_HEAD_H` need be undefined, use the directive `#undef` as

```
#undef _HEAD_H
```

## 4.3 Source Files and Linkages

### 4.3.1 *Separate Compilation*

A file is a traditional unit of storage in a file system and a traditional unit of compilation. A large program usually contains many files. When a file or something it depends on is changed, all of it has to be recompiled. Breaking a large program into suitable-size files can save tremendous compilation time in code development, since files (and those they have included) that have not been changed need not be recompiled.

When a source file is compiled, it is first preprocessed by a C++ preprocessor; that is, macros are substituted and `#include` directives bring in headers. The result from preprocessing a file is called a *translation unit*. Then the compiler works on this unit according to C++ language rules. Thus strictly speaking, there is a distinction between a source file and a translation unit, although most of the time these two names are used interchangeably.

To ensure successful separate compilation of the source files, the declarations in these files must be consistent. Note that an object can be declared many times (although it can only be defined once). After all source files have been compiled, their object codes will be linked by a *linker*, also called a *loader*. The linker can catch inconsistencies between separately compiled files. See §4.4.2 on how to compile and link different files on UNIX or Linux.

### 4.3.2 *External and Internal Linkages*

With separate compilation, some names may be declared or defined in one file and used in another. A name that can be used in translation units different from the one in which it is defined is said to have *external linkage*. For example, consider the following two files *file1.cc* and *file2.cc*.

```
// file1.cc
double x = 3.141;
double f(double y) { return y + 2.178; }

int main() { /* ... */ }
```

```
// file2.cc
extern double x;           // another file may define x
extern double f(double);   // another file may define f
void g(double z) { x = f(z); }
```

The variable  $x$  and function  $f()$  used in *file2.cc* are the ones defined in *file1.cc*. Thus  $x$  and  $f()$  are said to have external linkage. External variables have external linkage. Note that  $x$  and  $f$  are both global and external according to definitions in §2.1. A name that can be referred to only in the translation unit in which it is defined is said to have *internal linkage*. Thus the function  $g()$  in *file2.cc* has internal linkage and can not be used in *file1.cc*. The keyword *extern* in front of a function such as *double f(double)* on the second line in *file2.cc* can be omitted.

An object must be defined only once in a program, although it can be declared many times as long as the declarations agree exactly with each other. For example, consider the following two files.

```
// file3.cc
int i = 5;           // this declaration is also a definition
float a = 3.14;      // this defines a to be 3.14
extern char c;       // a declaration only

int main() { /* ... */ }
```

```
// file4.cc
int i = 10;          // i is defined again, illegal
extern double a;
extern char c;
char c = 'c';        // definition of c
```

There are two errors in these two files: the variable  $i$  is defined twice and  $a$  is declared twice with different types. Note  $c$  is declared three times but defined only once, which is not an error. These two errors can not be caught by the compiler since each file is compiled separately. They are linkage errors that should be detected by the linker.

At this moment, it may be a good idea to look at a few more examples of declarations and definitions:

```
double d;             // definition
int f(int);           // declaration
extern int i;         // declaration
struct s;             // declaration

namespace mv { int m = 100; } // definition
inline int g(int n) { return n; } // definition
const int j = 5;      // definition
```

```
extern const int k = 5;           // definition
typedef unsigned int Uint;       // definition
struct p2d { double x, y; };     // definition
```

The first statement is a definition since *d* is initialized by default. The names *f*, *i*, and *s* are declarations and must be defined somewhere. However, the names *mv*, *g*, *j*, *k*, *Uint*, and *p2d* are definitions since they introduce and define some entities into the program. Notice that *Uint* is just defined to be a synonym to *unsigned int* (it does not introduce a new type) and *k* is a constant that can be accessed in another file in which it is declared (using the statement *extern const int k*).

By default, *const*, *typedef*, and macros have internal linkage in C++. For example, it is legal (but confusing) to have the following two files.

```
// file5.cc
#define DEBUG 1                // internal linkage
const double d = 3.14;        // internal linkage
typedef unsigned int Uint;     // internal linkage
extern const int i = 100;      // external linkage

// file6.cc
#define DEBUG 0                // internal linkage
const double d = 2.17;        // internal linkage
typedef unsigned long int Uint; // internal linkage
extern const int i;            // declaration
```

However, an *inline* function must be defined in every file in which it is used with exactly the same definition. For example,

```
// file7.cc
inline int f(int i) { return i + i*i; }

// file8.cc
inline int f(int i) { return i + i*i; }
```

The compiler needs to know its definition in order to make code substitution. An *inline* function can not be made *extern* to simplify the implementation of C++ compilers.

These specifications may be very confusing and error-prone. The best way to handle them is to put them in a common header file and include it in different source files. An example is:

```
// file9.h          a header file
#include <iostream>    // include a standard library
#include <cmath>       // include a standard library

#define DEBUG 1      // user's stuff
const double d = 2.17;
```

```

extern long double pi;
typedef unsigned int Uint;
struct s { int i, j; };
enum color { red, green, blue};
namespace matvec { /* ... */ }

inline int f(int i) { return i + i*i; }

extern double g(long double, int i);

// file10.cc      for definitions of user's functions
#include "file9.h"

long double pi = std::atan(1)*4;

double g(long double d, int i) {      // definition of g
    // ...
    return d + i;
}

// file11.cc      a source file the main program
#include "file9.h"
int main() {
    double a = g(pi, 5) + f(10) + d;
    // ...
    std::cout << "a = " << a << '\n';
}

```

Here all common declarations and definitions are put in header file *file9.h* and source files *file10.cc* and *file11.cc* both include this header. A *struct*, *enum*, named *namespace*, *inline* function, and so on, must be defined exactly once in a program. Strictly speaking, this rule is violated when their definitions are put in a header file and included in different source files. For convenience, the language allows this since the definitions appear in different translation units and are exactly the same. See §4.2.2 for what should be and what should not be put in header files.

It is worthwhile to note that there are differences in C and C++ in terms of linkage. For example, consider these two files:

```

// file12.cc
float y;
float f() { return 5; }

int main() { g(); }

// file13.cc

```

```
float y;
void g() { y = f(); }
```

There are three errors as C++ programs: *y* is defined twice (the declarations of *y* are also definitions since the compiler initializes *y* to a default value) in the two files, and *f()* in *file13.cc* and *g()* in *file12.cc* are not declared. However, they are correct C programs.

### 4.3.3 Linkage to Other Languages

Sometimes a programmer has to deal with codes written in C++ and other languages such as C and FORTRAN. It can be difficult in general, since different languages may differ in the layout of function arguments put on the run-time stack, in the use of machine registers, in the layout of built-in types such as integers and arrays, and so on. For example, the layout of two-dimensional arrays is row by row in C++ and C, but column by column in FORTRAN, and arguments to functions are passed exclusively by reference in FORTRAN.

A user can specify a linkage convention in an *extern* declaration. For example,

```
extern "C" char* strcpy(char*, const char*);
                                // C linkage convention

extern "C" {                    // C linkage convention
    int f(int);                // group several declarations
    int g(int);                // in one block
    double sqrt(double);
}
```

Note that the letter C in double quotes following *extern* means to link these function calls according to the C linkage convention. Here C does not refer to a language but to a linkage convention. It can also be used to link FORTRAN and assembler codes that conform to C implementation conventions.

In general, C code can be mixed with C++ code statement by statement, provided the C code does not contain C++ keywords as user-defined names. This enables many C programs to be called directly from and mixed with C++ programs. Vaguely speaking, C is a subset of C++. However, dealing with a mixture of C++ and FORTRAN programs is platform-dependent. For example, a FORTRAN function name *sum()* could be compiled to a symbol name *sum*, *sum\_*, *sum\_\_*, or *SUM*, depending on the compiler. For examples on how to mix C/C++ and FORTRAN programs, see [LF93, Arn98].

## 4.4 Some Useful Tools

This section discusses a few useful tools (some in UNIX and Linux) for timing, debugging, and profiling a program, and for managing files and creating libraries.

### 4.4.1 How to Time a Program

C++ inherits from C the time and date library with header in `<ctime>` and `<time.h>`.

They have a function `clock()` that provides access to the underlying machine clock and returns an approximation to the number of CPU “clock ticks” used by the program up to the current time. The returned value can be divided by `CLOCKS_PER_SEC` to convert it into seconds. The macro `CLOCKS_PER_SEC` is defined in `<ctime>` or `<time.h>`, but the rate at which the clock time runs is machine-dependent. If the CPU clock is not available, the value `-1` is returned. The function `time()` returns the calendar time, normally expressed as the number of seconds elapsed since 1 January 1970. Other units and starting dates are possible. The prototypes of these two functions and related declarations are given in `<ctime>` and `<time.h>` and look like the following.

```
#define CLOCKS_PER_SEC 100          // machine dependent
typedef long clock_t;
typedef long time_t;

clock_t clock(void);                // return number of CPU clock ticks
time_t time(time_t* tp);           // return calendar time

double difftime(time_t t1, time_t t0); // return t1 - t0
char* ctime(const time_t *tp);       // convert calendar time to a string
```

When the variable `tp` in function `time(tp)` is not the null pointer, the returned value also gets stored in the object pointed to by `tp`. Note that the calendar time (wall time) and CPU clock time may not be the same in a time-shared operating system such as UNIX and Linux.

For example, the following program computes the wall times and CPU times for a number of multiplications in data types of *double* and *float*.

```
#include <iostream>                // for output
#include <time.h>                   // for measuring time

int main() {
    int n = 100000000;
    double d, dpi = 3.1415926535897932385;
```

```

float f, fpi = 3.1415926535897932385;
time_t tm0 = time(0);           // wall time at this point
clock_t ck0 = clock();          // clock ticks at this point

for (int i = 0; i < n; i++) d = (double(i) + dpi)*dpi;
time_t tm1 = time(0);           // wall time at this point
clock_t ck1 = clock();          // clock ticks at this point
cout << "wall time = " << difftime(tm1,tm0) << " seconds.\n";
cout << "CPU time = " << double(ck1 - ck0)/CLOCKS_PER_SEC
    << " seconds.\n";

for (int i = 0; i < n; i++) f = (float(i) + fpi)*fpi;
time_t tm2 = time(0);           // wall time at this point
clock_t ck2 = clock();          // clock ticks at this point
cout << "wall time = " << difftime(tm2,tm1) << " seconds.\n";
cout << "CPU time = " << double(ck2 - ck1)/CLOCKS_PER_SEC
    << " seconds.\n";

cout << "The current time is: " << ctime(&tm2) << "\n";
}

```

A run on a Pentium II PC running Linux (RedHat 5.1) with a GNU C++ compiler (gcc version egcs-2.91.66) produces the results:

```

wall time = 8 seconds.
CPU time = 7.23 seconds.
wall time = 4 seconds.
CPU time = 3.61 seconds.
The current time is: Sat Jun  3 15:56:45 2000

```

A run on an SGI workstation running UNIX (IRIX64) with a GNU C++ compiler (gcc version 2.8.1) produces the results:

```

wall time = 8 seconds.
CPU time = 7.69 seconds.
wall time = 10 seconds.
CPU time = 8.73 seconds.
The current time is: Fri Nov 12 14:16:39 1999

```

The computing speed generally depends on the underlying machine and compiler. It is surprising to observe that arithmetic in single precision may not be faster than in double precision on some machines. Some old C and C++ compilers promote *float* to *double* so that arithmetic in single precision can actually be slower than in double precision.

### 4.4.2 Compilation Options and Debuggers

When a program consists of different files, for example, *main.cc*, *pgm0.cc*, *pgm1.cc*, and *pgm2.cc*, the UNIX/Linux command with the `-c` option compiles them:

```
c++ -c main.cc pgm0.cc pgm1.cc pgm2.cc
```

If there are no errors, corresponding object files ending with *.o* will be created. Then the `-o` option links these object files to form a machine-executable object file:

```
c++ -o pgm main.o pgm0.o pgm1.o pgm2.o
```

If `-o pgm` is left out above, the executable object file will be created in the default file *a.out*, instead of *pgm*. These two commands can also be combined into one:

```
c++ -o pgm main.cc pgm0.cc pgm1.cc pgm2.cc
```

Below are some useful options for the C++ compiler.

<code>-c</code>	compile only, generate corresponding <i>.o</i> files
<code>-o name</code>	put executable object file in <i>name</i>
<code>-O</code>	attempt code optimization
<code>-g</code>	generate code suitable for the debugger
<code>-p</code>	generate code suitable for the profiler
<code>-I dir</code>	look for <i>#include</i> files in directory <i>dir</i>
<code>-D name=def</code>	place at the top of each source file the line <code>#define name def</code>
<code>-S</code>	generate assembler code in corresponding <i>.s</i> files
<code>-M</code>	create a makefile

Your compiler may not support all of these options and may provide others. Also your compiler may not be named *c++*.

A debugger allows the programmer to know the line number in a program where a run-time error has occurred, and to see the values of variables and expressions at each step. This can be very helpful in finding out why a program is not running successfully. The debuggers *gdb* and *dbx* are generally available on UNIX and Linux systems. The code must be compiled with the `-g` option. To use *gdb* on the object code *pgm*, for example, at the command line type

```
gdb
```

Then type *run pgm* (followed by the argument list if *pgm* has any) and it will run the program. Errors, if existing, will be displayed. Then typing *where* gives the exact line number and the function in which the error has occurred. Below are some useful commands.



run <i>pgm</i> [ <i>arglist</i> ]	run program <i>pgm</i> with argument list <i>arglist</i>
print <i>exp</i>	display the value of the expression <i>exp</i>
c	continue running the program after stopping
next	execute next program line after stopping
step	execute next line (step into the line)
help <i>name</i>	show information about GDB command <i>name</i>
bt	backtrace, display the program stack
quit	exit the debugger

The UNIX/Linux manpage may give more information on how to use it. See [LF93] for debuggers *dbx* and *xdb*, and other tools.

In UNIX, the *-p* option used in compiling a program produces code that counts the number of times a function is called. The profiler *prof* then generates an execution profile, which can be very useful when working to improve execution time efficiency. Look at the example:

```
#include <iostream>           // for output
#include <time.h>              // for measuring time
#include <stdlib.h>            // for pseudorandom number
#include <algorithm>          // for sort()

int main() {

    int n = 100000;
    double* dp = new double [n];

    srand(time(0));           // seed random number generator
    for (int i = 0; i < n; i++) dp[i] = rand()%1000;

    sort(dp, dp+n);           // sort array in increasing order
}
```

The function *srand()* seeds the random number generator and causes the sequence generated by repeated calls to *rand()* to start in a different place. The function *sort()* sorts a sequence in increasing order by default and is discussed further in §10.2.1.

Suppose the program is called *sortrand.cc*. Compile it with the *-p* option:

```
c++ -o sortrand -p sortrand.cc
```

The executable object code is written in file *sortrand*. Next run the program by issuing the command

```
sortrand
```

Then the command

```
prof sortrand
```

causes the following to be printed on the screen.

%time	cumsecs	#call	ms/call	name
57.5	1.04	1	1040.00	_main
21.5	1.43			mcount
16.0	1.72	100000	0.00	.rem
2.8	1.77	100000	0.00	.mul
2.2	1.81	100000	0.00	_rand
0.0	1.81	1	0.00	.udiv
0.0	1.81	1	0.00	.umul
0.0	1.81	1	0.00	_cfree
0.0	1.81	1	0.00	_exit
0.0	1.81	1	0.00	_free
0.0	1.81	1	0.00	_gettimeofday
0.0	1.81	3	0.00	_malloc
0.0	1.81	2	0.00	_on_exit
0.0	1.81	1	0.00	_profil
0.0	1.81	1	0.00	_sbrk
0.0	1.81	1	0.00	_srand
0.0	1.81	1	0.00	_time

This execution profile gives the number of times that a function is called, average number of milliseconds per call, and cumulative times in seconds. For example, the function *rand()* was invoked 100000 times, took 0.00 milliseconds per call, and took total 1.81 seconds, while the function *main()* was called once, took 1040.00 milliseconds per call, and took total 1.04 seconds.

#### 4.4.3 Creating a Library

The UNIX/Linux operating system provides a utility called the archiver to create and manage libraries. The name of the command is *ar*. By convention, UNIX library files end in *.a* (*.lib* in DOS). For example, the standard C++ library may be in the file *libstdc++.a* in the directory */usr/lib*. Issue the UNIX/Linux command

```
ar t /usr/lib/libstdc++.a
```

The key *t* will display the titles (names) of the files in the library.

A programmer may create his or her own library. Suppose there are files *matvec.cc*, *mesh.cc*, and *fem.cc* that contain matrix-vector operation, mesh generation, and finite element analysis codes. To create a library, first compile these source files and obtain the corresponding *.o* files. Then type the two commands:

```
ar ruv libmvmf.a matvec.o mesh.o fem.o
```

```
ranlib libmvmf.a
```

The keys *ruv* in the first command stand for replace, update, and verbose, respectively. The library *libmvmf.a* is then created if it does not already exist. If it exists, then the *.o* files replace those of the same name already existing in the library. If any of these *.o* files are not already in the library, they are added to it. The *ranlib* command simply randomizes the library that is useful for the loader (linker).

If there is a program consisting of two files: *main.cc* and *iofile.cc*, and these two files call functions in the library *libmvmf.a*, then the library can be made available to the compiler by issuing the following command.

```
c++ -o pgm main.cc iofile.cc libmvmf.a
```

The functions that are invoked in *main.cc* and *iofile.cc* but not defined there will be searched for first in *libmvmf.a* and then in C++ standard libraries. Only those functions of *libmvmf.a* that are used in *main.cc* and *iofile.cc* will be loaded into the final executable file *pgm*. Typing all the commands can be tedious when there are a lot of files and libraries. A UNIX utility named *make* can achieve this by specifying all the commands in a file called *Makefile*; see §4.4.4.

#### 4.4.4 *Makefile*

The *make* utility (available always in UNIX and Linux and often in DOS as well) provides easy management of compiling source files and linking their object files, and convenient access to libraries and associated header files. It is efficient to keep a moderate or large size program in several files and compile them separately. If one of the source files needs to be changed, only that file has to be recompiled. The use of the *make* utility can greatly facilitate the maintenance of the source files in a program written in C++, C, FORTRAN, Java, and others.

The *make* command reads a file with a default name *makefile* or *Makefile*. The file contains dependency lines (e.g., how an object file depends on a source file and some header files) and action lines (e.g., how a source file is to be compiled with what options). As an example, let us write a *Makefile* for a program that contains a header file *matvec.h* and two source files *main.cc* and *matvec.cc*, all in the current directory. The file *main.cc* contains the function *main()* that calls some functions defined in *matvec.cc*. The header *matvec.h* is included in both *main.cc* and *matvec.cc*. The *Makefile* for such a simple program can be just as

```
mat: main.o matvec.o          # mat depends on main.o,matvec.o
    c++ -o mat main.o matvec.o  # link object code mat

main.o: main.cc matvec.h      # main.o rely on main.cc,matvec.h
    cc -c -O main.cc          # compile main.cc and optimize
```

```
matvec.o: matvec.cc matvec.h # what matvec.o depends on ?
    cc -c -O matvec.cc      # compile with option -O
```

A comment in *Makefile* begins with symbol `#` and extends to the rest of the line (just like `//` in a C++ program). The first line is a dependency line that indicates that *mat* depends on object files *main.o* and *matvec.o*. The second line is an action line that indicates how the object files are to be linked and the resulting executable object code is written in a file named *mat*. A dependency line must start in column 1 while an action line must begin with a tab character. A dependency line and the action lines following it are called a rule. So there are three rules *mat*, *main.o*, and *matvec.o* in this *Makefile*. Note that the name of a rule must be followed by a colon. Then the UNIX/Linux command

```
make mat
```

makes the rule *mat* in *Makefile*, which depends on two other rules *main.o* and *matvec.o*. These two rules will be made first according to their action lines. That is, the source files *main.cc* and *matvec.cc* are to be compiled with the `-O` option. Then the rule *mat* will cause the object code *main.o* and *matvec.o* to be linked and the final executable object code to be written in file *mat*.

If the source file *main.cc* is to be changed, but *matvec.h* and *matvec.cc* remain unchanged, then the command *make mat* will only cause the file *main.cc* to be recompiled and leave the third rule *matvec.o* alone. However, if the header file *matvec.h* is to be changed, then *main.cc* and *matvec.cc* will be recompiled since *main.o* and *matvec.o* both depend on *matvec.h* by the second and third rules.

Similarly, the command *make main.o* will cause the rule *main.o* to be made; in this case only *main.cc* will be compiled. By default, the command *make* without an argument simply makes the first rule in the *Makefile*. For this *Makefile*, the commands *make* and *make mat* are equivalent.

Certain rules and macros are built into *make*. For example, each *.o* file depends on the corresponding *.c* file and the second rule in the above *Makefile* is equivalent to

```
main.o: main.cc matvec.h    # main.o rely on main.cc, matvec.h
    cc -c -O $*.cc         # compile and attempt to optimize
```

where `$*.cc` expands to *main.cc* when *main.o* is being made. It stands for all *.cc* files on the dependency line. The system also echoes a message on the screen about the action. To suppress the message, the symbol `@` can be used:

```
main.o: main.cc matvec.h    # main.o rely on main.cc, matvec.h
    @cc -c -O $*.cc         # compile main.cc with option -O
```

A user can also define macros in a *makefile*. The following is a *makefile* that I use for compiling my code in files *main.cc*, *fem.cc*, and *fem.h* that call functions in libraries *matveclib.a* and *meshlib.a* and include headers *matvec.h* and *mesh.h* in two different directories.

```
# Makefile for source files fem.cc and main.cc that
# include fem.h. Sources main.cc and fem.cc call functions
# in my libraries matveclib.a and meshlib.a
# fem.h includes two headers matvec.h and mesh.h

# a macro BASE for my home directory.
# Value of the macro can be accessed by $BASE
BASE = /home/yang

# macro for name of my compiler
CC = c++

# macro for compilation options
CFLAGS = -O

# directories for include files matvec.h and mesh.h
# which are included in fem.h
INCLS = -I$(BASE)/c++/matvec -I$(BASE)/c++/mesh

# paths for my libraries matveclib.a and meshlib.a
# '\' means a continued line
LIBS = $(BASE)/c++/matvec/matveclib.a \
      $(BASE)/c++/mesh/meshlib.a

# name of executable file
EFILE = $(BASE)/c++/fem/xx

HDRS = fem.h
OBJS = main.o fem.o

fem: $(OBJS)
    @echo "linking ..."
    $(CC) $(CFLAGS) -o $(EFILE) $(OBJS) $(LIBS)

$(OBJS): $(HDRS)
    $(CC) $(CFLAGS) $(INCLS) -c $.cc

clean:
    rm -f $(OBJS) xx
```

```
# main.o depends on main.cc and fem.o depends on fem.cc
main.o: main.cc
fem.o: fem.cc
```

The files *main.cc* and *fem.cc* include the header file *fem.h*, all of which are in the same directory */home/yang/c++/fem* as the *Makefile*. Furthermore, *fem.h* includes two header files *matvec.h* and *mesh.h*, which reside, respectively, in two different directories:

```
/home/yang/c++/matvec
/home/yang/c++/mesh
```

The macro *INCLS* means to search for header files in these directories. When *main.cc* and *fem.cc* are compiled, these header files are searched for by the system. When *main.o* and *fem.o* are linked to generate the executable object file *xx*, the libraries *matveclib.a* and *meshlib.a* are loaded through the macro *LIBS*. Note that paths for C++ standard libraries such as *<cmath>* and *<complex>* and their header files need not be indicated since the system should find them automatically. Only paths for user-defined libraries and header files have to be indicated.

At UNIX or Linux command, type

```
make
```

It will compile *main.cc* and *fem.cc* and link *main.o* and *fem.o* into the executable object code *xx*. The last two lines in this *Makefile* indicate that *main.o* depends also on *main.cc* besides *fem.h* and *fem.o* also on *fem.cc*. Thus any change to *main.cc* will cause only *main.o* to be regenerated. The command *make clean* will remove the files defined in macro *OBJS* and the file *xx*.

UNIX also provides a utility called *SCCS* (Source Code Control System) for managing different versions of source code (or any other text files); see [LF93].

## 4.5 Standard Library on Strings

The C++ standard library *<string>* provides a type called *string* to manipulate character strings. Only a few examples are presented here to illustrate its use. Details can be found in [Str97, LL98].

### 4.5.1 Declarations and Initializations

A *string* can be initialized by a sequence of characters, by another string, or by a C-style string. However, it can not be initialized by a character or an integer. For example,

```

#include <string>
using namespace std;
int main() {
    string s0;           // empty string
    string s00 = "";     // empty string
    string s1 = "Hello"; // initialization
    string s2 = "world";
    string s3 = s2;
    string s33(5,'A');   // 5 copies of 'A', s33 = "AAAAA"

    string s333 = 'A';   // error, no conversion from char
    string s3333 = 8;    // error, no conversion from int

    s3 = 'B';           // OK, assign a char to string
}

```

Notice that assigning a *char* to a string is allowed, although it is illegal to initialize a string by a *char*.

#### 4.5.2 Operations

Certain arithmetic and operations are defined for *string*. For example, addition of strings means concatenation. The operator `+=` can be used to append to the end of a string. Strings can be compared to each other and to string literals. The function call `s.empty()` returns *true* if string *s* is empty and *false* otherwise, and `s.max_size()` gives the maximum size that a string can have. The function calls `s.size()` and `s.length()` both give the size of a string *s*. The size of a *string* must be strictly less than `string::npos`, which is the largest possible value for string size (it is equal to 4294967295 on one machine). Elements in a string *s* can be accessed by the subscripting operator `[]` (index changes from 0 up to `s.size() - 1`). For example,

```

if (s2 == s1 ) {           // test for equality of s1, s2
    s2 += '\n';           // append '\n' to end of s2
} else if (s2 == "world") {
    s3 = s1 + ", " + s2 + "!\n"; // s3 = "Hello, world!\n"
}

s3[12] = '?';             // assign '?' to s3[12]
char h = s3[12];         // h = '?'

int i = s3.length();      // size of s3, i = 14
int j = s3.size();        // size of s3, j = 14

```

The function call `s.substr(n,m)` gives a substring of *s* consisting of *m* characters starting from *s[n]*, `s.replace(n,m,t)` replaces *m* characters

starting from  $s[n]$  by string  $t$ ,  $s.insert(n, t)$  inserts a string  $t$  at  $s[n]$ , and  $s.find(t)$  returns the position of substring  $t$  in string  $s$  and returns  $npos$  if  $t$  is not a substring of  $s$ . Note that if  $npos$  is used as a character position, an exception *range\_error* will be thrown (see Chapter 9). They can be used as

```
// now s3 = "Hello, world?\n"
string s4 = s3.substr(7,5); // s4 = "world";
s3.replace(0,5,"Hi");      // s3 = "Hi, world?"
s3.insert(0,"Hi");         // insert "Hi" at s3[0]
unsigned int pos = s3.find("wor");
                        // find position of substring "wor"
unsigned int pos2 = s3.find("HW");
                        // pos2 = 4294967295 on one machine
s3.erase(0,2);            // erase two characters from s3
```

A related function, *rfind()* searches for a substring from the end of a string. The function *s.erase(n, m)* erases  $m$  characters from  $s$  starting from  $s[n]$ .

The usual comparison operators  $=$ ,  $!=$ ,  $>$ ,  $<$ ,  $>=$ , and  $<=$  are provided based on lexicographical ordering. In addition, many standard algorithms can be applied to strings; see §10.2.

### 4.5.3 C-Style Strings

Conversion from *string* to C-style strings (*char\**) is provided. For example,

```
const char* p = s3.c_str(); // convert to C-style string
const char* q = s3.data();  // convert to an array of chars
char c = q[0];              // c = first character in q

char* s6 = "Hello";
string s7 = s6;              // from C-style string to string
```

The function *s.data()* converts a *string*  $s$  to an array of characters while *s.c\_str()* produces a C-style string (*char\** or *const char\**).

Note that a C++ *string* is not an array of characters. That is, for a *string*  $s$ ,  $\&s[0]$  is different from  $s$ .

### 4.5.4 Input and Output

Strings can be read from and written to the screen or a file using the input and output operators. For example,

```
string s10;
cout << "Enter your city and state names\n";
cin >> s10;
cout << "Hi, " << s10 << "!\n";
```



If you type in your city and state as

```
Detroit Michigan
```

the output on the screen is *Hi, Detroit!* By default, a whitespace character terminates the input. Thus the second word *Michigan* above is not read to string *s10*. To read the whole line, the function *getline()* can be used:

```
cout << "Enter your name\n";
getline(cin, s10);           // read a whole line from cin
cout << "Hi, " << s10 << "\n";
```

The function

```
getline(istream istm, string strg, char eol);
```

reads a line from an input stream *istm* terminated by character *eol* to the string *strg*. If no *eol* argument is provided, a newline character '\n' is used as the delimiter.

#### 4.5.5 C Library on Strings

The header `<cstring>` or `<string.h>` contains functions operating on C-style strings. For example,

```
char* strcpy(char*p, const char*q);
           // copy from q into p, including terminator
char* strcat(char*p, const char*q);
           // append from q to p, including terminator
char* strncpy(char*p, const char*q, int n);
           // copy n chars from q into p
char* strncat(char*p, const char*q, int n);
           // append n chars from q into p

size_t strlen(const char* p);           // length of p
int strcmp(const char*p, const char*q); // compare p, q
int strncmp(const char*p, const char*q, int n);
           // compare first n characters in p and q
```

The pointer argument above is assumed to be nonzero, and the array of *char* it points to is assumed to be terminated by 0. The *strn*-functions pad with 0 if there are not *n* characters to copy. The string comparisons return 0 if the strings are equal, a negative number if the first string is lexicographically before the second, and a positive number otherwise. The type *size\_t* is an unsigned integral type defined in the standard headers `<stddef.h>` and `<stdlib.h>`.

The header `<cctype>` or `<ctype.h>` contains functions dealing with ASCII characters. For example,

```

int isalpha(int);      // Is 'a'..'z' 'A'..'Z' ?
int isupper(int);      // Is 'A'..'Z' ?
int islower(int);      // Is 'a'..'z' ?
int isdigit(int);      // Is digit '0'..'9' ?
int isspace(int);      // Is space ' ' '\t' '\v' ?
int isalnum(int);      // isalpha() | isdigit()

int toupper(int);      // to uppercase
int tolower(int);      // to lowercase

```

These functions take *int* as argument since a character is treated as a small integer in C and C++. Equivalent functions for wide characters can be found in `<cwtype>` and `<wtype.h>`. The type *wchar\_t* for wide characters is defined in `<stddef.h>` and `<cstdint.h>`.

The header `<cstdio>` or `<stdio.h>` contains a function called *sprintf()* which can concatenate strings and numeric numbers together:

```

char s11[50];
int ii = 5;
int jj = 80;
sprintf(s11, "%s%d%s%d%s", "ex", ii, "h", jj, "\n");
cout << s11;          // ex5h80 and a newline are printed out

```

In the specification `"%s%d%s%d%s"` above, *s* means a string and *d* means an integer (a digit). This function is useful when we want to generate a string for an output file name, which represents, for instance, the example number and grid size such as *ex5h80*; see §4.6.4 and §4.6.6.

## 4.6 Standard Library on Streams

The C++ standard libraries `<iostream>` and `<fstream>` provide functions for reading from and writing to the screen or files, which are illustrated through examples in this section. More details can be found in [Str97, LL98].

### 4.6.1 Formatted Integer Output

The function *setf()* can set the format in which an integer is output. For example,

```

int main() {
    int i = 123456789;
    cout << i << " " << i << '\n';          // decimal
    cout.setf(ios_base::oct, ios_base::basefield); // octal
    cout << i << " " << i << '\n';          // print i in octal
}

```

```

    cout.setf(ios_base::hex, ios_base::basefield); // hex
    cout << i << " " << i << '\n';           // print i in hex
    cout.setf(ios_base::dec, ios_base::basefield); // decimal
    cout << i << " " << i << '\n';
}

```

The identifiers *ios\_base*, *ios\_base::dec* (decimal), *ios\_base::oct* (octal, base 8), *ios\_base::hex* (hexadecimal, base 16), and *ios\_base::basefield* (which base in the output?) are defined in `<iostream>`. Since the name *ios*, instead of *ios\_base*, was used in prestandard C++, many standard C++ compilers may continue to support *ios*. This program will output the value of *i* in decimal, octal, hexadecimal, and decimal formats:

```

123456789 123456789
726746425 726746425
75bcd15 75bcd15
123456789 123456789

```

Setting *showbase* will enable one to see which base is used. By default, a number is decimal. An octal number is preceded by 0, while a hexadecimal number is preceded by 0x. For example, the program segment

```

int i = 123456789;
cout.setf(ios_base::showbase); // show base of output
cout << i << " " << i << '\n'; // decimal, default
cout.setf(ios_base::oct, ios_base::basefield);
cout << i << " " << i << '\n'; // octal, base 8
cout.setf(ios_base::hex, ios_base::basefield);
cout << i << " " << i << '\n'; // hex, base =16

cout.setf(0, ios_base::basefield); // reset to default
cout << i << " " << i << '\n';

```

will output to the screen:

```

123456789 123456789
0726746425 0726746425
0x75bcd15 0x75bcd15
123456789 123456789

```

Setting *ios\_base::basefield* to 0 resets the base to default, that is, decimal. This technique can also print out an octal or hexadecimal integer in decimal. For example, let *i* = 012345 or *i* = 0x12345 above. Setting *showbase* shows the base of an integer. Similarly, setting *showpoint* prints trailing zeros and setting *showpos* prints explicitly + for positive integers. Once set, these format flags remain valid until they are unset by calling the function *unsetf()*, as in *cout.unsetf(ios\_base::showbase)*. To set or unset several flags in one statement, the bitwise or operator can be used as in *cout.unsetf(ios\_base::showbase | ios\_base::showpos)*.

### 4.6.2 Formatted Floating Point Output

Floating point output is controlled by a format and a precision:

- The *general* format presents a number that best preserves its value in the space available; the precision specifies the maximum number of digits.
- The *scientific* format presents a number with one digit before the decimal point and an exponent; the precision specifies the maximum number of digits after the decimal point.
- The *fixed* format presents a number with an integral part and a fractional part; the precision specifies the maximum number of digits after the decimal point.

The *general* format is the default and the default precision is six digits. The function *setf()* can set the format and *precision()* can set the precision. For example,

```
double d = 12345.6789;
cout << d << " " << d << '\n';    // default
cout.setf(ios_base::scientific, ios_base::floatfield);
cout << d << " " << d << '\n';    // scientific
cout.setf(ios_base::fixed, ios_base::floatfield);
cout << d << " " << d << '\n';    // fixed
cout.setf(0, ios_base::floatfield); // reset
cout << d << " " << d << '\n';    // to default
```

outputs

```
12345.7  12345.7
1.234568e+04  1.234568e+04
12345.678900  12345.678900
12345.7  12345.7
```

The identifiers *ios\_base::scientific* (for scientific format), *ios\_base::fixed* (for fixed format), and *ios\_base::floatfield* are defined in `<iostream>`. Setting *ios\_base::floatfield* to 0 resets the format back to the default. Note that the default precision of six digits is applied to the whole number of the general format, while only to the fractional part of scientific and fixed formats. To increase the accuracy of an output, the *precision()* function can be used. For example,

```
double d = 12345.67890123456789;
cout.setf(ios_base::scientific, ios_base::floatfield);
cout.setf(ios_base::uppercase); // E in scientific format
cout.precision(15);             // precision = 15
cout << d << " " << 1000*d << '\n';
```

```
cout.precision(10);                // precision = 10
cout << d << " " << 1000*d << '\n';
```

The output is:

```
1.234567890123457E+04 1.234567890123457E+07
1.2345678901E+04 1.2345678901E+07
```

Since the output is in the scientific format, the precision means the number of digits after the decimal point. Notice that *E*, instead of *e*, appears in the exponential part of the scientific format by setting *ios\_base::uppercase*.

### 4.6.3 Output Width

The function *width()* can control the number of characters that the next output number or string occupies. If the number is not long enough, whitespace by default or some other “padding” character specified by the function *fill()* will be used. For example,

```
double d = 12345.678987654321;
cout.width(25);           // output width 25 chars
cout.precision(15);
cout << d << '\n';
cout.width(25);           // output width 25 chars
cout.precision(8);
cout.fill('#');           // use '#' for padding extra space
cout << d << '\n';
```

It will produce the following output, which is aligned to the right by default:

```
12345.6789876543
#####12345.679
```

Adjustment of the alignment can be made by setting *adjustfield* to *left* (left alignment), *internal* (internal alignment, put fill characters between sign and value), or *right* (right alignment). For example,

```
double d = - 12345.678987654321;
cout.precision(15);
cout.setf(ios_base::left, ios_base::adjustfield);
cout.width(25);
cout << d << '\n';
```

The alignment is adjusted to the left in this output.

The *width()* function sets the minimum number of characters. If the number has more digits, more characters will be output in order not to lose accuracy. The field width can be reset to its default by calling *cout.width(0)*.

#### 4.6.4 Input and Output Files

The C++ header `<fstream>` provides functions for inputting from and outputting to files. In particular, they have types *ifstream* for input file stream, *ofstream* for output file stream, and *fstream* for both input and output. The open mode of a file can be

- *in* : to open for reading.
- *out* : to open for writing.
- *app* : to append.
- *binary* : to open in binary mode, rather than in text mode.
- *ate* : to open and seek to end of file.
- *trunc* : to truncate a file to 0-length.

For example, the declarations

```
ifstream infile("data", ios_base::in);
                        // open file for input
ofstream outfile(result, ios_base::out);
                        // open file for output
```

declare the variable *infile* to be an input file stream that opens the file *data* for reading, and *outfile* to be an output file stream that opens a file whose name is stored in string *result* for writing.

The following is a complete program that reads an integer and a double from a file called *data*, generates a string for an output file name, and writes to the file.

```
#include <stdio.h>           // for sprintf()
#include <fstream>           // for input/output files

int main() {
    ifstream infile("data", ios_base::in); // open file to read
    int i;
    double d;
    infile >> i;                // read from "data"
    infile >> d;

    char result[20];             // output file name
    sprintf(result, "%s%d%s%d%s", "ex", i, "-", int(1/d), "h");
    ofstream outfile(result, ios_base::out); // open output file
    outfile << "example number: " << i << '\n';
    outfile << "grid size: " << d << '\n';
}
```

```

infile.close();           // close input file stream
outfile.close();          // close output file stream
}

```

A file needs to be closed by calling function *close()* when it is no longer used. There is a maximum number of files that can be opened at a time. If the input file *data* has only two lines:

```

5
0.01

```

then the program first reads the first number 5 to variable *i* and then reads the second number 0.01 to variable *d*. The function *sprintf()* causes the variable *result* to have value *ex5-99h*. The output of the program is written in a file called “ex5-99h”:

```

example number: 5
grid size: 0.01

```

The bitwise or operator `|` can be used to choose a file to have more than one open mode. For example, to open a file for both writing and appending, or to open a file for both writing and reading, we can do:

```

ofstream wfile("data2", ios_base::out | ios_base::app);
                        // open file for write and append
fstream rfile("data3", ios_base::in | ios_base::out);
                        // open file for read and write

```

To generate formatted output to files, the functions *setf()*, *precision()*, and *width()* can be applied to file streams in the same way as described earlier for *cout*. For example,

```

ofstream ouf("res", ios_base::out); // open output file
ouf.setf(ios_base::scientific, ios_base::floatfield);
ouf.precision(10);
ouf.width(30);
ouf << 12345.67890 << '\n';

```

Finally, here is an example on how to read a file with comments. Suppose there is a file called *data4* that contains a comment line as the first line and some comments preceded by a slash sign `/` on every following line:

```

data4: input file for a test example
4           / example number
0.01        / grid size

```

The input file with comments is more readable since it indicates that 4 represents an example number and 0.01 a grid size. The program, which must ignore the first line and everything after the slash sign on other lines, may be written as

```

#include <string>
#include <stdlib.h>
#include <fstream>

int main(int argc, char* argv[]) {

    if (argc != 3)
        cout << "input and output files must be specified\n";

    ifstream infile(argv[1], ios_base::in);
    if (!infile)
        cout << "can not open file: " << argv[1] << " for read\n";
    string s;
    int i;
    double d;

    infile.ignore(80, '\n'); // ignore the first line

    getline(infile, s, '/'); // get a line terminated by '/'
    i = atoi(s.c_str()); // convert to an int
    infile.ignore(80, '\n'); // ignore rest of line, comments

    getline(infile, s, '/'); // get next line terminated by '/'
    d = atof(s.c_str()); // convert to a float

    ofstream outfile(argv[2], ios_base::out);
    if (!outfile)
        cout << "can not open file: " << argv[2] << " for write\n";
    outfile.setf(ios_base::scientific, ios_base::floatfield);
    outfile << "example number: " << i << '\n';
    outfile << "grid size:      " << d << '\n';

    infile.close();
    outfile.close();
}

```

The function *sm.ignore(streamsize n, int ct)* skips from an input stream *sm* at most *n* characters terminated by the character *ct*, where *streamsize* is an integral type for the size of a stream, possibly *int*. The default terminating character for *ignore()* is *end-of-file*, represented by *EOF*, which is defined in *<iostream>*. The default number of characters ignored is one. Thus a call *sm.ignore()* without any arguments means to throw the next character away. Compile this program and suppose the object code is written in *a.out*. Then the UNIX/Linux command

```
a.out data4 result
```



will read from file *data4* and write the output into a file called *result*.

#### 4.6.5 Input and Output of Characters

There are many operations defined for reading and writing characters. For example, the function call

```
ism.read(Ch* p, streamsize n)
```

reads at most *n* characters (including whitespace and newline characters) from input stream *ism* (of type *istream*) and stores them in *p*[0], *p*[1], ..., where *Ch* can be *char*, *unsigned char*, or *signed char*, and *streamsize* is an integral type for the size of a stream, typically *int*. It returns a reference of type *basic\_istream* to the input stream object to which it is applied. The function *ignore()* is similar to *read()* except that it does not store the characters read but simply ignores them. The function call *ism.gcount()* returns the number of bytes read from *ism* in the last call *read()*. Similarly, the function call

```
osm.write(const Ch* p, streamsize n)
```

writes at most *n* characters (including whitespace and newline characters) stored in *p*[0], *p*[1], ..., into output stream *osm*. It returns a reference of type *basic\_ostream* to the output stream object to which it is applied. These two functions have similar return types to the input and output operators of *Cmpx* in §6.1. The types *basic\_istream* and *basic\_ostream* are defined in *<iostream>*. The two functions can be used to write a program that reads from an input stream and changes every character into its corresponding upper case:

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

main() {
    ifstream infile("readfile", ios_base::in);
    char c;
    int n = 0, m = 0;
    while ( infile.read(&c, 1) ) { // read 1 char at a time
        switch(c) {
            case ',': n++; break;
            case '\n': m++; break;
        }
        c = toupper(c);
        cout.write(&c, 1);          // write c on screen
    }
    cout << "There are " << n << " commas and "
```

```

        << m << " lines in the input file.\n";
    }

```

It also counts the number of commas and newlines in the input stream. Suppose a file named *readfile* contains the following.

```

The function read(p, n) reads at most n characters and
stores them in a character array p[0], p[1], up to
p[n - 1]. The returned object will be converted to false
when end of file is encountered.

```

Then the program outputs the following on the screen:

```

THE FUNCTION READ(P, N) READS AT MOST N CHARACTERS AND
STORES THEM IN A CHARACTER ARRAY P[0], P[1], UP TO
P[N - 1]. THE RETURNED OBJECT WILL BE CONVERTED TO FALSE
WHEN END OF FILE IS ENCOUNTERED.

```

There are 3 commas and 4 lines in the input file.

The functions *read()* and *write()* may read or write many characters at a time. When only one character is read or written, the functions *get(Ch&)* and *put(Ch)* are better suited. Using *get()* and *put()*, the program above can be rewritten as

```

main() {
    ifstream infile("readfile", ios_base::in);
    char c;
    int n = 0, m = 0;
    while ( infile.get(c) ) {        // read 1 char at a time
        switch(c) {
            case ',': n++; break;
            case '\n': m++; break;
        }
        c = toupper(c);
        cout.put(c);                // write c on screen
    }
    cout << "There are " << n << " commas and "
         << m << " lines in the input file.\n";
}

```

The function *get()* is overloaded. Calling *get()* without any arguments returns the character just read.

Another version of *get()* is:

```

ism.get(Ch* p, streamsize n, Ch delimiter = '\n');

```

which reads at most  $n - 1$  characters from input stream *ism* and places them into  $p[0]$ ,  $p[1]$ ,  $\dots$ ,  $p[n - 2]$ . The third argument *delimiter* is a character stating that the reading of characters should be terminated when it

is encountered. The default delimiter character is `'\n'`. The terminating character itself is not read into array *p*. Thus the user may have to remove the next character before performing a second read. The pointer *p* must point to an array of at least *n* characters, since *get()* places 0 at the end of the characters (C-style strings). The function call

```
ism.getline(Ch* p, streamsize n, Ch delimiter = '\n');
```

is very similar to *get()*, except that it discards the delimiter rather than leaving it as the next character to be read in the input stream. The call to *gcount()* returns the number of characters actually extracted by the last call of *get()* or *getline()*. Here is an example using *get()* and *gcount()* to read a file and count the number of characters on each line and the total number of lines in the file:

```
void f(istream& ism) {
    int n = 0;
    char line[1024];
    while ( ism.get(line, 1024, '\n') ) {
        n++;
        cout << "there are " << ism.gcount() << " characters "
              << "on line " << n << '\n';
        ism.ignore();      // ignore terminate character: '\n'
    }
    cout << "There are " << n << " lines in input file.\n";
}

main() {
    ifstream infe("readfile", ios::in);
    f(infe);
}
```

Without ignoring the newline character on each line (through the function call *ism.ignore()*; see §4.6.4), this would cause an infinite loop. The output of this program on the input file *readfile* above is:

```
there are 56 characters on line 1
there are 54 characters on line 2
there are 57 characters on line 3
there are 34 characters on line 4
There are 4 lines in input file.
```

Notice the newline character on each line is not read by *get()*. The function *f()* above can also be alternatively written using *getline()*:

```
void f(istream& ism) {
    int n = 0;
    char line[1024];
```

```

while ( ism.getline(line, 1024, '\n') ) {
    n++;
    cout << "there are " << ism.gcount() << " characters "
        << "on line " << n << '\n';
}
cout << "There are " << n << " lines in input file.\n";
}

```

Thus the function *getline()* is normally preferred over *get()* when one line needs to be read at a time. Notice this function is different from *getline()* discussed in §4.6.4. The output of this program on *readfile* is:

```

there are 57 characters on line 1
there are 55 characters on line 2
there are 58 characters on line 3
there are 35 characters on line 4
There are 4 lines in input file.

```

The newline character on each line of the file *readfile* is also read by *getline()*.

There are three other *istream* operators that *puts back*, *ungets*, or *peeks* a character in an input stream:

```

ism.putback(Ch c); // put character c back into stream ism
ism.unget();       // put most recently read character back
ism.peek();        // return next character (or EOF)
                  // but do not extract it from ism

```

The following program illustrates how they might be used.

```

void f(istream& ism) {
    char c, d;
    while (ism.get(c)) {
        switch(c) {
            case '/': // look for comments starting with //
                if (ism.peek() == '/') // if there are two slashes,
                    ism.ignore(1024, '\n'); // ignore rest of the line
                break;
            case '<': // look for operator <<
                d = ism.get();
                if (d != '<') ism.putback(d); // put d back
                break;
            case '&': // look for operator &&
                if (ism.get() != '&') ism.unget();
                break;
        }
    }
}

```

### 4.6.6 String Streams

The standard library `<sstream>` contains two types *istream* (for input string streams) and *ostream* (for output string streams), which can be used to read from a *string* and write to a *string* using input and output streams.

To illustrate how *ostream* can be used, consider a file named *data* that contains only two lines:

```
5
0.01
```

A program can be written that reads the two numbers from this file, generates a string such as "ex5-99h" from the two numbers just read, opens a file with this string as its name, and finally writes some information to this file:

```
#include <string>
#include <fstream>
#include <sstream>
using namespace std;

main() {
    ifstream infe("data", ios_base::in);
    int i;
    double d;
    infe >> i;
    infe >> d;                // read from input file

    ostringstream tmp;        // declare variable tmp
    tmp << "ex" << i << "-" << int(1/d) << "h";
    string ofname = tmp.str(); // convert it into string

    ofstream otfe(ofname.c_str(), ios_base::out);
    otfe << "example number : " << i << '\n';
    otfe << "grid size: " << d << '\n';

    infe.close();
    otfe.close();
}
```

It declares *tmp* to be a variable of type *stringstream* and writes some numeric values and characters into *tmp* using the standard output operator. The function call *tmp.str()* converts an *stringstream* object *tmp* into a C++ *string*. Then *c\_str()* is called to convert a C++ *string* into a C string (*char\**), which is used as the output file name. Notice that an *stringstream* object can be initialized and the statements

```
ostreamstream tmp;           // declare variable tmp
tmp << "ex" << i << "-" << int(1/d) << "h";
```

are equivalent to:

```
ostreamstream tmp("ex");     // initialize variable tmp
tmp << i << "-" << int(1/d) << "h";
```

Compiling and running this program produces a file named *ex5-99h*:

```
example number: 5
grid size: 0.01
```

This is exactly what was achieved in §4.6.4 using the C function *sprintf()*.

An *istream* is an input stream reading from a *string*. It can be used to convert a string (with whitespace characters) into different parts, some of which may have numeric values. For example,

```
main() {
    string s = "ex 5 - 9.9 h";
    istringstream ism(s);      // initialize ism from string s

    int i;
    double d;
    string u, v, w;
    ism >> u >> i >> v >> d >> w;      // i = 5, d = 9.9
}
```

This achieves more than what C functions *atoi()*, *atof()*, and *atol()* do.

The libraries *<string>* and *<sstream>* are very convenient to use when manipulating strings. Compared to C-style strings (*char\**), the user does not need to do memory allocation and deallocation. However, they may cause performance problems when used extensively, since they are built on the top of C-style strings.

## 4.7 Iterative Methods for Nonlinear Equations

In this section, the bisection and Newton's methods are presented to iteratively find a solution to a nonlinear equation  $f(x) = 0$  in a given interval  $[a, b]$ . If  $f$  is a continuous function on  $[a, b]$  and  $f(a)$  and  $f(b)$  have different signs (one is positive and the other is negative), then the Intermediate Value Theorem guarantees that there is at least one root  $r$  in  $(a, b)$  such that  $f(r) = 0$ . The goal of an iterative method is to find a sequence of numbers  $\{r_n\}$  such that  $r_n$  converges to a root  $r$  when  $n \rightarrow \infty$ . On a computer, an iterative process has to be stopped after finite many iterations. The stopping criterion can be  $|r_m - r_{m-1}| < \delta$  or  $|f(r_m)| < \epsilon$ , for some given small numbers  $\delta$  and  $\epsilon$ . When either criterion is satisfied,  $r_m$  may

be taken as an approximation to an exact root  $r$ . Due to the approximate nature of an iterative method and computer roundoff errors,  $r_m$  may never satisfy  $f(r_m) = 0$  exactly, no matter how many iterations are used. The value  $f(r_m)$  is called a residual.

#### 4.7.1 Bisection Method

The bisection method assumes that  $f$  is continuous on  $[a, b]$  and  $f(a)f(b) < 0$ . It first computes the middle point  $c = (a + b)/2$  and then tests if  $f(a)f(c) < 0$ . If it is true, then  $f$  must have a root in the smaller interval  $[a, c]$ . If it is false, then  $f$  must have a root in  $[c, b]$ . In either case, rename the smaller interval as  $[a, b]$ , which contains a root but whose size is reduced by half. Repeat this process until  $|b - a|$  or  $|f(c)|$  is small, and then the middle point  $c$  is taken as an approximate root of  $f$ . This algorithm can be written easily in a recursive program as

```
double bisctn0(double a, double b, double (*f)(double),
               double delta, double epsn) {
    double c = (a + b)*0.5;           // middle point
    if (fabs(b-a)*0.5 < delta || fabs(f(c)) < epsn) return c;
    (f(a)*f(c) < 0) ? b = c : a = c;
    return bisctn0(a, b, f, delta, epsn);
}
```

Note that the function can also be alternatively declared as

```
typedef double (*pfn)(double);       // define a function type
double bisctn0(double a, double b, pfn f,
               double delta, double epsn);
```

With the help of the function pointer  $pfn$ , this equivalent declaration may be more readable to C++ beginners. Also the ternary operator  $? :$  is used to make the code look concise. It may also be equivalently written using an *if*-statement:

```
if (f(a)*f(c) < 0)
    b = c;
else
    a = c;
```

Observe the close resemblance of the code to the algorithm. It recursively calls the function itself to shrink the interval size by half in each recursion and stops when the interval size or residual is small.

Now this program can be tested to find solutions to:

- (a)  $f(x) = x^{-1} - 2^x$ ,  $[a, b] = [0, 1]$ ,
- (b)  $f(x) = 2^{-x} + e^x + 2 \cos x - 6$ ,  $[a, b] = [1, 3]$ ,

$$(c) \ f(x) = (x^3 + 4x^2 + 3x + 5)/(2x^3 - 9x^2 + 18x - 2), \quad [a, b] = [0, 4].$$

Common declarations and definitions can be put in a header file called *ch4it.h*:

```
// file ch4it.h
#include <iostream>           // for input and output
#include <math.h>             // for math functions
#include <stdlib.h>           // for exit()

const double delta = 1.0e-8;
const double epsn = 1.0e-9;

double bisctn0(double, double, double (*)(double),
               double, double);
double fa(double x);
double fb(double x);
double fc(double x);
```

The source code containing function definitions can be put in another file called *ch4it.cc*:

```
// file ch4it.cc
#include "ch4it.h"           // include header file

double bisctn0(double a, double b, double (*)(double),
               double delta, double epsn) {
    // ... put its definition here
}

double fa(double x) {        // test function (a)
    if (x) return 1.0/x - pow(2,x);
    else {
        cout << "division by zero occurred in function fa().";
        exit(1);
    }
}

double fb(double x) {        // test function (b)
    return pow(2,-x) + exp(x) + 2*cos(x) - 6;
}

double fc(double x) {        // test function (c)
    double denorm = ((2*x-9)*x + 18)*x - 2; // nested multiply
    if (denorm) return (((x+4)*x + 3)*x + 5)/denorm;
    else {
        cout << "division by zero occurred in function fc().";
    }
}
```



```

    exit(1);
}
}

```

The functions  $fa()$  and  $fc()$  may involve division by zero. When it occurs, the program is terminated by calling the function *exit()*, declared in header *<stdlib.h>*.

A main program can be written to find the roots of the functions:

```

// file mainch4it.cc
#include "ch4it.h"
int main() {
    // find a root of fa()
    double root = bisctn0(1.0e-2, 1, fa, delta, epsn);
    cout << "Approximate root of fa() by bisctn0() is: "
        << root << '\n';
    cout << "Fcn value at approx root (residual) is: "
        << fa(root) << '\n';

    // find a root of fb()
    root = bisctn0(1, 3, fb, delta, epsn);
    cout << "\nApproximate root of fb() by bisctn0() is: "
        << root << '\n';
    cout << "Fcn value at approx root (residual) is: "
        << fb(root) << '\n';

    // find a root of fc()
    root = bisctn0(0, 4, fc, delta, epsn);
    cout << "\nApproximate root of fc() by bisctn0() is: "
        << root << '\n';
    cout << "Fcn value at approximate root (residual) is:"
        << fc(root) << '\n';
}

```

Compiling and running the program gives the following output.

```

Approximate root of fa() by bisctn0() is: 0.641186
Fcn value at approx root (residual) is: -1.48037e-09

```

```

Approximate root of fb() by bisctn0() is: 1.82938
Fcn value at approx root (residual) is: 9.4115e-09

```

```

Approximate root of fc() by bisctn0() is: 0.117877
Fcn value at approximate root (residual) is: 2.24202e+09

```

Observe that the program finds approximate roots for functions  $fa()$  and  $fb()$  with very small residuals. However, the residual for  $fc()$  is so large!

A closer look reveals that  $f(c)$  does not have a root in the given interval, although it changes sign at the endpoints (why?). It is a good idea to check the correctness of computer output whenever possible. In the case of finding a solution to an equation, a small residual normally implies a good approximate answer.

The function `bisctn0()` resembles the algorithm **closely and has worked fine** on three test problems. However, it is **not very efficient and robust** in the following sense. **First, it** invokes three function evaluations:  $f(a)$  once and  $f(c)$  twice, in each recursion. **Second,** the product  $f(a) * f(c)$  may cause overflow or underflow while  $f(a)$  and  $f(c)$  are within range. **Third,** the middle point  $c$  can be updated in a more robust way as  $c = a + (b-a)/2$ . There are examples in which the middle point computed by  $c = (a+b)/2$  moves outside the interval  $[a, b]$  on a computer with limited finite precision. It is numerically more robust to compute a quantity by adding a small correction to a previous approximation. An improved version can be defined as

```
double bisctn1(double a, double b, double (*f)(double),
               double u, double delta, double epsn) {
    double e = (b - a)*0.5;      // shrink interval size
    double c = a + e;            // middle point
    double w = f(c);             // fcn value at middle point

    if (fabs(e) < delta || fabs(w) < epsn) return c;
    ((u>0 && w<0) || (u<0 && w>0)) ? (b=c):(a=c, u=w);
    return bisctn1(a, b, f, u, delta, epsn);
}
```

This version `bisctn1()` requires only one function evaluation  $f(c)$  per recursion ( **$u = f(a)$**  is passed as an argument to the next recursion) and avoids the unnecessary product  $f(a) * f(c)$ . However, these two functions `bisctn0()` and `bisctn1()` have a common disadvantage. That is, **both can lead to an infinite recursion** (why?). A safeguard is to put a limit on the number of recursions. From §3.8.9, a static local variable may be used to count the number of recursions and lead to the next version:

```
double bisctn2(double a, double b, double (*f)(double),
               double u, double delta, double epsn,
               int maxit) {
    static int itern = 1;
    double e = (b - a)*0.5;      // shrink interval size
    double c = a + e;            // middle point
    double w = f(c);             // fcn value at middle point

    if (fabs(e)<delta || fabs(w)<epsn || itern++ > maxit)
        return c;
```

```

    ((u>0 && w<0) || (u<0 && w>0)) ? (b=c) : (a=c, u=w);
    return bisctn2(a, b, f, u, delta, epsn, maxit);
}

```

This new version *bisctn2()* achieves the goal that it stops when the number of recursions is larger than a prescribed number *maxit*, but has a side effect. Since static local variable *itern* persists in all calls to *bisctn2()*, it may cause the function to terminate prematurely in subsequent calls. For example,

```

double root = bisctn2(1.0e-2, 1, fa, fa(1.0e-2), 1.0e-8,
                     1.0e-9, 40);
root = bisctn2(1, 3, fb, fb(1), 1.0e-8, 1.0e-9, 40);
root = bisctn2(0, 4, fc, fc(0), 1.0e-8, 1.0e-9, 40);

```

The first call on *fa* takes 27 recursions to stop. The static local variable *itern* = 27 at the beginning of the second call on *fb*, which is terminated prematurely after 14 recursions when *itern* = 41. The third call on *fc* is terminated after only the first recursion.

To overcome this difficulty, one more argument can be passed (by reference) to the function to count the number of iterations. It can be written as

```

double bisectionr(double a, double b, double (*f)(double),
                 double u, double delta, double epsn,
                 int maxit, int& itern) {
/*****
  bisection algm: recursive version, returns an approximate
                  root of a fcn in a given interval
  a, b: endpoints of interval in which a root lies
  f:    function is defined in interval [a,b] or [b,a].
  u:    = f(a)
  delta: root tolerance, return when updated interval is
          narrower than it
  epsn:  residual tolerance, return when residual is
          smaller than it
  maxit: maximum number of iterations allowed
  itern: iteration count, it must be initialized to 1.
*****/

  double e = (b - a)*0.5;    // shrink interval size
  double c = a + e;          // middle point
  double w = f(c);           // fcn value at middle point

  if (fabs(e)<delta || fabs(w)<epsn || itern++ > maxit)
    return c;
  ((u>0 && w<0) || (u<0 && w>0)) ? (b=c) : (a=c, u=w);
}

```

```
    return bisectionr(a,b,f,u,delta,epsn,maxit,itern);
}
```

The reference variable *itern* must be initialized to 1 before each call and stores the number of iterations after a call is finished. For example,

```
int itrn = 1;
double root = bisectionr(1.0e-2, 1, fa, fa(1.0e-2),
                        1e-8, 1e-9, 500, itrn);
cout << "Number of iterations used = " << itrn << '\n';

itrn = 1;          // itrn must be initialized to 1
root = bisectionr(1, 3, fb, fb(1), 1e-8, 1e-9, 500, itrn);
cout << "Number of iterations used = " << itrn << '\n';
```

Run this program to know that the first call takes 27 iterations and the second call takes 28 iterations to stop. They produce the same approximate roots as *bisctn0()*.

The procedure above is generally called an *incremental approach* to software development. It starts with a simple version representing the main idea of an algorithm. From there, more efficient and robust versions can be built.

A nonrecursive version of the bisection algorithm can now be easily coded as

```
double bisection(double a, double b, double (*f)(double),
                double delta, double epsn, int maxit) {
/*****
bisection algm: nonrecursive version, returns approximate
                root of a fcn in a given interval
a, b:  endpoints of interval in which a root lies
f:     function is defined in interval [a,b] or [b,a].
delta: root tolerance, return when updated interval is
        narrower than it
epsn:  residual tolerance, return when residual is
        smaller than it
maxit: maximum number of iterations allowed
*****/

    double u = f(a);          // fcn value at left pt
    double e = b - a;         // interval length
    double c;                  // store middle point

    for (int k = 1; k <= maxit; k++) { // main iteration
        e *= 0.5;              // shrink interval by half
        c = a + e;             // update middle pt
        double w = f(c);       // fcn value at middle pt
```

```

    if (fabs(e) < delta || fabs(w) < epsn) return c;
    ((u>0 && w<0) || (u<0 && w>0)) ? (b=c) : (a=c, u=w);
}
return c;
} // end bisection()

```

All the disadvantages in early recursive versions *bisctn0()*, *bisctn1()*, and *bisctn2()* are avoided in this nonrecursive version. Both *bisectionr()* and *bisection()* provide safeguards against entering infinite loops. This is a good consideration since roundoff errors and numeric instability can lead to unpredictable behaviors.

#### 4.7.2 Newton's Method

Let  $f(x)$  be a function of independent variable  $x$ ,  $f'(x)$  its first-order derivative, and an initial approximation  $x_p$  to a root  $r$  of  $f$ . Newton's algorithm tries to find a better approximation  $x_n$  to this root  $r$ , and repeat this process until convergence. The idea is to first compute the root of its linear approximation at  $x_p$  and then let  $x_n$  be this root. The linear approximation of  $f(x)$  at  $x_p$  is:

$$L_f(x_p) = f(x_p) + f'(x_p)(x - x_p).$$

This is exactly the first two terms of its Taylor's expansion at  $x_p$ . Geometrically, the curve  $f(x)$  is approximated near the point  $(x_p, f(x_p))$  by its tangent line at this point. To find the root of this linear approximation, set  $L_f(x_p) = 0$ , that is,

$$f(x_p) + f'(x_p)(x - x_p) = 0,$$

and solve for  $x$  to get  $x = x_p - f'(x_p)^{-1}f(x_p)$ . Then let this  $x$  be the next approximate root  $x_n$ . That is,

$$x_n = x_p - f'(x_p)^{-1}f(x_p).$$

This is the iterative formula of Newton's method. If  $|x_n - x_p| < \delta$  or  $|f(x_n)| < \epsilon$ , where  $\epsilon$  and  $\delta$  are two given small positive numbers, then stop and take  $x_n$  to be the approximate root of  $f(x)$  near  $r$ . Otherwise let  $x_p = x_n$  and repeat this process. In order to prevent entering an infinite loop, this process should also be stopped when the total number of iterations has exceeded a prescribed number. It can be coded into a C++ program as

```

typedef double (*pfn)(double);    // define a function type

double newton(double xp, pfn f, pfn fd, double delta,
              double epsn, int mxt) {
/*****

```

```

newton's algorithm: finds an approximate root of  $f(x) = 0$ 
xp:    an initial guess of a root
f:     the function whose root is to be found
fd:    the derivative function of f
mxt:   maximum number of iterations allowed
delta: program stops when distance of two iterates is < it
epsn:  program stops when residual is less than it.
*****/
double v = f(xp);           // fcn value at initial guess
double xnew;                // store new iterate

for (int k = 1; k <= mxt; k++) { // main iteration loop
    double derv = fd(xp);      // derivative at xp
    if (!derv) {
        cout << "Division by 0 occurred in newton().\n";
        exit(1);              // stop if divisor == 0
    }

    xnew = xp - v/derv;        // compute new iterate
    v = f(xnew);              // fcn value at new iterate
    if (fabs(xnew - xp) < delta || fabs(v) < epsn) return xnew;
    xp = xnew;
}                             // end main iteration loop
return xnew;
} // end newton()

```

This function can be tested on a problem of finding a root of  $f(x) = x^3 - 5x^2 + 3x + 7$  near 5. Compiling and running the program (with appropriate declarations and header files),

```

double f(double x) {
    return ((x - 5)*x + 3)*x + 7; // nested multiply
}

double fder(double x) {
    return (3*x - 10)*x + 3;
}

int main() {
    double root = newton(5, f, fder, 1e-8, 1e-9, 500);
    cout << "Approx root near 5 by newton method is: "
        << root << '\n';
    cout << "Fcn value at approximate root (residual) is: "
        << f(root) << '\n';
}

```

gives the following output

```
Approx root near 5 by newton method is: 3.65544
Fcn value at approximate root (residual) is: 3.71647e-14
```

Thus Newton's method finds an approximate root with very small residual to this polynomial function. In general, the initial approximation  $x_p$  (also called initial guess) may not have to be very close to a root  $r$ . But a good initial guess can speed up the convergence and even affect the convergence or divergence of the method. That is, Newton's method may converge with one initial guess and diverge with another. Once a good initial guess is obtained, Newton's method converges more rapidly than the bisection method.

If  $f(r) = 0$  and  $f'(r) \neq 0$ , then  $r$  is called a simple root of  $f$ . Newton's method is designed for finding a simple root due to the following observation. If  $f'(r) \approx 0$  and  $x_p$  is an approximation to  $r$ , then the iterative formula of Newton's method would involve a very small number or zero in the denominator.

Let the second derivative  $f''$  be continuous and  $r$  a simple root of  $f$ . Then there exists a neighborhood of  $r$  and a constant  $C$  such that if  $x_p$  is in the neighborhood, the next iterate  $x_n$  is closer to  $r$  and

$$|r - x_n| \leq C|r - x_p|^2.$$

See [KC96] for a proof. Such a convergence is called *quadratic*.

The root-finding functions can be put in a namespace. The common declarations are put in a header file and object codes in a library. A user can just include the header file and call the library for root-finding applications.

Notice that the bisection and Newton's methods are programmed in traditional C or FORTRAN style in this section. They require passing a function pointer as argument, which is hard to be inlined or optimized, and may impose function calling overhead for each function evaluation  $f(x)$  inside them. In §7.7 some efficient techniques are applied to overcome the function calling overhead in passing a function pointer to another function call.

## 4.8 Exercises

- 4.8.1. Implement matrix and vector namespaces as outlined in §4.1. Add functions for vector addition, vector-scalar multiplication, matrix addition, and matrix-vector multiplication. Compute one, two, and maximum norms of vector  $v = (v_i)_{i=0}^{n-1}$  and one, maximum, and Frobenius norms of matrix  $m = (a_{i,j})_{i,j=0}^{n-1}$ , and the matrix-vector product  $m \times v$ , where  $v_i = \sin(i)$  and  $a_{i,j} = \cos(i^2 + j)$ , for different matrix and vector sizes  $n = 100, 500$ , and  $1000$ . Manage the files by using a Makefile if you are on a UNIX/Linux system.

- 4.8.2. Implement a matrix-vector library including vector and matrix norms, and additions and multiplications in single, double, and long double precisions. Put common declarations and type definitions in a header file, implementations in a source file, and the main program in another source file. Test your program on the matrix  $m$  and vector  $v$  defined in Exercise 4.8.1 with different sizes and precisions.

Note that three versions of the same functions in single, double, and long double precisions seem to be tedious and hard to maintain. When the implementation is to be changed, for example, the straightforward evaluation of 2-norms is changed to a more robust evaluation (see Exercise 4.8.5), then every version of the function has to be changed. This has been done in a routine way in languages such as FORTRAN 77 and 90, and C. Using templates one version suffices; see Chapter 7.

- 4.8.3. Modify the program in §4.4.1 and compare the computing speeds in double, single, and extended double precisions on your machine.
- 4.8.4. Compute the golden mean (§2.4) in *float*, *double*, and *long double*. Write the results in a file with formatted output (e.g., scientific format, 30-character output width, 20 digits of precision, aligned to the left).
- 4.8.5. To derive a robust vector 2-norm formula for a vector  $v = [v_0, v_1, \dots, v_{n-1}]$ , that will not cause overflow if  $\|v\|_2$  is within range, let  $r_0 = 0$  and  $r_{i+1}^2 = r_i^2 + v_i^2$ , for  $i = 0, 1, 2, \dots, n-1$ . That is,

$$r_1^2 = v_0^2, \quad r_2^2 = v_0^2 + v_1^2, \quad \dots, \quad r_n^2 = v_0^2 + v_1^2 + \dots + v_{n-1}^2.$$

Then  $\|v\|_2 = \sqrt{r_n^2} = r_n$ . From the recursive relation  $r_{i+1}^2 = r_i^2 + v_i^2$ , to avoid possible overflow in summing the squares  $r_i^2 + v_i^2$ , compute  $r_{i+1} = r_i \sqrt{1 + (v_i/r_i)^2}$  when  $r_i$  is large and compute  $r_{i+1} = |v_i| \sqrt{1 + (r_i/v_i)^2}$  when  $|v_i|$  is large. Thus an alternative but more robust definition for `Vec::twonorm()` is:

```
double Vec::twonorm(const double* const v, int size) {
    if (size > maxsize) cout << "vector size too large.\n";
    double norm = fabs(v[0]);
    for (int i = 1; i < size; i++) {
        double avi = fabs(v[i]);
        if (norm < 100 && avi < 100)
            norm = sqrt(norm*norm + avi*avi);
        else if (norm > avi) norm *= sqrt(1 + pow(avi/norm, 2));
        else norm = avi*sqrt(1 + pow(norm/avi, 2));
    }
    return norm;
}
```



This robust definition involves more operations and should be slower than the straightforward evaluation as defined in §4.1. Write a program that compares the time efficiency of these two definitions, on a vector  $v = (v_i)_{i=0}^{n-1}$ , where  $v_i = \cos(i)$  and  $n = 900000$ . Adjust the value of  $n$  if your computer takes too long or too short. A robust algorithm is achieved sometimes at the expense of efficiency.

- 4.8.6. Using the idea of Exercise 4.8.5, the function `Mat::frobnorm()` (see §4.1) can be defined in a more robust way:

```
double Mat::frobnorm(const double** const a, int size) {
    double norm = Vec::twonorm(a[0], size); // 2-norm of row 0
    for (int i = 1; i < size; i++) {
        double avi = Vec::twonorm(a[i], size);
        if (norm < 100 && avi < 100)
            norm = sqrt(norm*norm + avi*avi);
        else if (norm > avi) norm *= sqrt(1+pow(avi/norm,2));
        else norm = avi*sqrt(1 + pow(norm/avi,2));
    }
    return norm;
}
```

It can also be defined without using the function call `Vec::twonorm()`, which leads to a possibly more efficient version:

```
double Mat::frobnorm(const double** const a, int size) {
    double norm = 0;
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            double avi = fabs(a[i][j]);
            if (norm < 100 && avi < 100)
                norm = sqrt(norm*norm + avi*avi);
            else if (norm > avi)
                norm *= sqrt(1 + pow(avi/norm,2));
            else norm = avi*sqrt(1 + pow(norm/avi,2));
        }
    }
    return norm;
}
```

Write a program that compares the time efficiency of the three versions of the matrix Frobenius norm function, on an  $n \times n$  matrix evaluated repeatedly  $m$  times. You may take  $n = 1000$  and  $m = 100$ . Adjust the values of  $n$  and  $m$  to fit your computer.

- 4.8.7. Time the program computing the first 40 Fibonacci numbers in §2.4 and the recursive program in Exercise 3.14.15. To clearly see the time efficiency (or inefficiency) of recursive and nonrecursive function calls, invoke each function inside a loop for  $m$  times, where  $m$  is a large integer (e.g.,  $m = 20000$ ). Output the results directly into a file. Write your own Makefile for compiling and managing the programs if you are using UNIX or Linux.
- 4.8.8. Write a function that takes a decimal *int* as argument and prints out (to the screen and a file) the integer in hexadecimal, octal, and binary formats.
- 4.8.9. Call all the functions (defined in §4.7.1) for the bisection method inside a *main()* program to find a root of
- (a)  $f(x) = x^{-1} - \tan x$ ,  $[a, b] = [0, 2]$ ,  
 (b)  $f(x) = x - e^{-x}$ ,  $[a, b] = [1, 2]$ .

Comment on the advantages and disadvantages of these methods.

- 4.8.10. Apply Newton's method to find a root of the function  $f(x) = 1 - x - e^{-x}$  near  $x = 1$ , and a root of  $g(x) = \tan x - x$  near  $x = 4.5$  (test what you get if starting from  $x = 5$  instead).
- 4.8.11. Amounts  $A_1, A_2, \dots, A_n$  are placed in an investment account (e.g., a bank savings account) at the beginning of month 1, 2,  $\dots$ ,  $n$ , respectively. Let  $B_n$  be the balance in the account just after amount  $A_n$  has been invested at the end of month  $n$ . The average annual return rate  $r$  (assuming monthly interest compounding) of this investment account is a root to the equation:

$$B_n = \sum_{i=1}^n A_i \left(1 + \frac{r}{12}\right)^{n-i}.$$

Write a function that computes the average annual return rate  $r$ , given  $B_n, A_1, A_2, \dots$ , and  $A_n$ . If a man invests 100 dollars each month for the first year, 200, 300, 400, and 500 dollars each month for the second, third, fourth, and fifth years, respectively, and has balance 28000 dollars right after the 60th investment, what is his average annual return rate (assuming monthly compounding)? Hint: Newton's or the bisection method may be used to find the root and Horner's algorithm should be used in calculating the summation.

- 4.8.12. Amounts  $A_1, A_2, \dots, A_n$  are placed in an investment account (e.g., a mutual fund account) at the beginning of month 1, 2,  $\dots$ ,  $n$ , respectively. Let  $B_n$  be the balance in the account just after amount  $A_n$  has been invested at the end of month  $n$ . The average annual return rate

$r$  (assuming continuous interest compounding; it is very close to daily compounding) of this investment account is a root to the equation:

$$B_n = \sum_{i=1}^n A_i e^{r(n-i)/12}.$$

Write a function that computes the average annual return rate  $r$ , given  $B_n$ ,  $A_1$ ,  $A_2$ ,  $\dots$ , and  $A_n$ .

- 4.8.13. The library function *system()* in *<stdlib.h>* provides access to operating system commands. A string can be passed to it as the argument and this string is treated as an operating system command. For example, the program

```
#include <stdlib.h>
int main() {
    system("date");
}
```

executes the command *date* that causes the current date to be output to the screen. Write a program on UNIX/Linux that lists all files with a suffix *.cc* using the following statements.

```
char command[30];
sprintf(command, "ls %s", "*.cc"); // include <stdio.h>
system(command);                  // include <stdlib.h>
```

Can you write a program that moves every *.cc* file into a corresponding *.cpp* file?

A related function is *asm()*, whose argument is a string literal representing a piece of assembly code that will be inserted into the generated code at the place where it is specified.



# 5

## Classes

The C++ construct *class* provides an encapsulation mechanism so that a user can encapsulate data and functions manipulating the data together to define new types, which can be used as conveniently as built-in types. This can also increase the readability and maintainability of a program, and separate implementation details from its user interface.

Starting with this chapter, advanced features of C++ on object-oriented programming and generic programming are introduced. However, in this chapter, only basic concepts about *class* are discussed. This chapter ends with a section on numeric methods for ordinary differential equations.

### 5.1 Class Declarations and Definitions

A *class* is a user-defined type that may contain data members and function members that manipulate the data. There can also be functions, called friends, which are not members of the class but have access to members of the class. A class may have a *private* part that may be accessed only by members and friends of the class, and a *public* part that can be accessed freely in the program. A member of a class is *private* by default and can also be explicitly declared *private*. The words *class*, *private*, *public*, and *friend* are reserved words. Thus a class provides certain access restriction for some of its members and achieves encapsulation and information hiding.

Objects of a class are created and initialized by its member functions specifically declared for this purpose; such functions are called *constructors*.

A constructor is recognized by having the same name as the class itself. A member function can be specifically declared to “clean up” objects of a class when they are destroyed; such a function is called a **destructor**.

As an example, consider defining *pt2d* to be a *class* with *x* and *y* coordinates of a two-dimensional point as data members and a few functions manipulating the data:

```
class pt2d {                                // class for 2D points
private:                                   // private members
    double x;                             // x coordinate
    double y;                             // y coordinate
public:                                    // public members
    pt2d(double r, double s) {             // constructor
        x = r; y = s;
    }

    void move(double, double);              // move to new position
    void move(pt2d);                        // move to new position
    void draw() const;                     // a const member
    friend double norm(pt2d);               // a friend
};
```

表示该函数不会改变类的成员变量值

Notice a semicolon is needed after a class declaration. A class declaration is actually a class definition since it defines a new type. The members *x* and *y* are private and can only be accessed by member functions *pt2d::pt2d()*, *pt2d::move()*, *pt2d::draw()*, and friend *norm()*, where the function *pt2d::pt2d()* is defined inside the class declaration and others are defined below. The member *pt2d::pt2d()*, whose name is identical to the name of the class, is called a *constructor* and all objects of class *pt2d* must be created by the constructor. For example, the statements

```
pt2d p(5.4, 6);                            // call constructor
pt2d q(5, 0);                              // call constructor
```

create an object called *p* of type *pt2d* with *x*-coordinate 5.4 and *y*-coordinate 6, and an object *q* with coordinates *x* = 5 and *y* = 0. A destructor is not necessary for this class since the constructor does not allocate space for it using the operator *new*. Many examples in later sections need destructors; see, for example, §5.2. Classes are different from other types in that an object of a class must be constructed through a constructor. In this sense, objects of a class are different from other objects.

The class scope operator *::* must be used for definitions of members outside a class declaration. A member function such as *pt2d::draw()*, declared using the *const* suffix, can only read but can not modify members of the class. The keyword *const* here is part of the type and must not be omitted in the definition when defined outside the class declaration. In contrast, nonconstant member functions imply that they modify members of the

C++中，new这种方式只适合与指针，与C#和Java不一样

class and should not be invoked on constant class objects. Definitions of other function members and the friend can be:

```
inline void pt2d::move(double r, double s) {
    x = r; y = s;           // move point to (r, s)
}

inline void pt2d::move(pt2d p) {
    x = p.x; y = p.y;       // move point to p
}

inline void pt2d::draw() const {    // draw coordinates
    cout << '(' << x << ', ' << y << ')';    // const member
}

inline double norm(pt2d p) {        // friend definition
    return sqrt(p.x*p.x + p.y*p.y); // distance to origin
}
```

Notice the difference between the ways that members are accessed by a function member such as *move()* and a friend such as *norm()*. In *move()*, data members *x* and *y* can be used directly (they refer to members of an object for which the member function is called), while the class qualifier must be used in *norm()* such as *p.x* and *p.y*. Defining *move()* to be a member function and *norm()* a friend here is only to show the difference on how to define member functions and friends. The function *norm()* may also be alternatively defined as a member and *move()* as friends. A friend does not belong to a class but has access to members of the class. More details on friends are given in §5.3. Member functions defined inside a class declaration are *inline* by default. They can also be alternatively made *inline* outside the class declaration like *pt2d :: move()* above. Again, *inline* is only a suggestion to the compiler for code substitution to avoid function calling overhead. Short member functions and friends can be made *inline* for possible run-time efficiency. Long definitions should be put outside a class declaration to make the interface look clean.

The following main program defines a few objects of class *pt2d* and calls member functions and friend to show how this class can be used.

```
int main() {
    pt2d a(0, 0);           // create object a = (0,0)
    pt2d b(5, 0);           // create object b = (5, 0)
    pt2d c(3, -40);         // create object c = (3, -40)

    a.move(23, -3.5);       // move a to (23, -3.5)
    b.move(c);              // move b to (3, -40)
    c.draw();               // draw point c
}
```

```
double dist = norm(c);    // distance from c to origin
}
```

In the above, member functions *move()* and *draw()* are called using the member selection operator *.* following a class object, while a friend *norm()* is called in the same way as an ordinary function.

The following statements in *f()* are illegal.

So, usually, define  
a void constructor

```
void f(pt2d a) {
    double ax = a.x;    // can not access private member
    pt2d d;              // error, initializer missing
}
```

That is, private members of a class may only be accessed by its member functions and friends as member *x* can be used in member function *move()* and friend *norm()*, but not in other functions such as *main()* and *f()* above to prevent them from accidental modification or misuse. Since the constructor of *pt2d* requires two arguments, an object can not be constructed (for *d* in *f()* above) without specifying them. The protection of private members in a class leads to a safer user-defined type that is also easier to maintain. If the representation of a class is to be changed, only its member functions and friends need be changed. User code depends only on the public interface and need not be changed.

It is sometimes convenient to provide several constructors; the compiler can choose the correct one to use according to the type of its arguments just as in function overloading. For example, the class *pt2d* can be defined to have three constructors:

```
class pt2d {
public:
    pt2d() { x = y = 0; }    // in addition to other members
    pt2d(double p) { x = p; y = 0; }    // default constructor
    pt2d(double p, double q) { x = p; y = q; }
};
```

Now objects of *pt2d* can be more conveniently constructed as

```
pt2d a;                // a = (0,0), default constructor
pt2d b(5);             // b = (5, 0)
pt2d c(3, -40);        // c = (3, -40)
```

1) overloading  
2) default value

A constructor that takes no arguments is called *a default constructor*. Thus the object *a* above is constructed by the default constructor of class *pt2d*. These three constructors can also be combined into one:

```
class pt2d {
public:
    pt2d(double p = 0, double q = 0) { x = p; y = q; }
};
```



This is nothing but providing default arguments to a function.

An object of a class can be used to initialize another object and assigned to another object. For example,

```
pt2d b(5);           // b = (5, 0)
pt2d c(3, -40);      // c = (3, -40)
pt2d d = b;          // initialize d to b
c = d;               // assign d to c
```

In the above, *d* is constructed by a memberwise copy from object *b*, which is called copy construction or copy initialization. The assignment *c = d* causes the content of *d* to be assigned to *c* by a memberwise copy; such an assignment is called copy assignment. Copy construction and copy assignment are different operations in that a copy construction creates a new object while a copy assignment potentially modifies an object already created.

By definition, a *struct* is a class with all members public. For example,

```
struct pt3d {
    double x, y, z;           // all members are public
};
```

The members *x*, *y*, and *z* are public by default and can be accessed by any function. Constructors, destructor, and other member functions may also be defined for it. When a class does not provide a constructor, the compiler tries to generate a default constructor for it to use. For example, in the statement

```
pt3d p;
```

the compiler creates an object *p* of type *pt3d* and initializes the members *x*, *y*, and *z* to some default value. But there are classes for which the compiler can not generate a default constructor; see §5.6.


The keyword this refers to a constant pointer that points to the object for which a member function is invoked. For a nonconstant member function of class *X*, the pointer *this* is of type *X\* const*, and for a constant member function, *this* has type *const X\* const*. For example, the member functions *move()* and *draw()* can also be implemented as

```
inline void pt2d::move(double r, double s) {
    this->x = r;  this->y = s;           // move point to (r, s)
}

inline void pt2d::draw() const {       // draw coordinates
    cout << '(' << this->x << ', ' << this->y << ')';
}
```

Using *this* when referring to members is usually unnecessary. Its main use is for defining member functions that manipulate pointers directly. Useful examples are given in §5.2 and the next chapter.

As another example, we now define a *class* for definite integrals that contains two data members for the lower and upper integral bounds, another data member as a pointer to a function for the integrand, and a few function members and a friend for manipulating and evaluating the definite integral such as the Trapezoidal Rule and Simpson's Rule. Numeric integration was discussed in §3.13 in the procedural programming style as in C and FORTRAN 90. Now *class* can be used to achieve encapsulation of data and functions and information hiding. It can be declared as



```
typedef double (*pfn)(double); // define a function pointer

class integral {                // members by default are private
    double lower;               // lower integral bound
    double upper;              // upper integral bound
    pfn integrand;             // integrand function
public:                         // public members
    integral(double a, double b, pfn f){ // a constructor
        lower = a; upper = b; integrand = f;
    }
    double lowbd() const { return lower; } // const fcn member
    double upbd() const { return upper; }  // const fcn member
    void changebd(double, double);         // nonconst member
    double trapezoidal(int) const;         // const fcn member
    friend double simpson(integral, int);  // a friend
};                                         // note semicolon
```

The members *lower*, *upper*, and *integrand* are private and can be accessed only by its function members such as *integral::lowbd()* and friends such as *simpson()*. Thus public functions *integral::lowbd()* and *integral::upbd()* are provided to return the lower and upper integral bounds. When a class has a constructor, all objects of the class must be constructed by calling a constructor. For example, the declaration

```
integral di(0, 5.5, sqrt);
```

declares *di* to be of type *integral* and creates an object with initialization *lower* = 0.0, *upper* = 5.5, and *integrand* = *sqrt*, where *sqrt()* is the square-root function in *<math.h>*. A destructor is not needed for the class since the user does not explicitly allocate space for it using the operator *new*.

Definitions of other function members and the friend can be:

```
inline void integral::changebd(double a, double b) {
    lower = a;                // change integral bounds to a, b
    upper = b;
}

double integral::trapezoidal(int n) const { // const member
```

```

double h = (upper - lower)/n;          // size of subinterval
double sum = integrand(lower)*0.5;
for (int i = 1; i < n; i++) sum += integrand(lower + i*h);
sum += integrand(upper)*0.5;
return sum*h;
}

```

```

double simpson(integral ig, int n) {      // a friend
    double h = (ig.upper - ig.lower)/n;
    double sum = ig.integrand(ig.lower)*0.5;
    for (int i = 1; i < n; i++)
        sum += ig.integrand(ig.lower + i*h);
    sum += ig.integrand(ig.upper)*0.5;

    double summid = 0.0;
    for (int i = 1; i <= n; i++)
        summid += ig.integrand(ig.lower + (i-0.5)*h);

    return (sum + 2*summid)*h/3.0;
}

```

The following main program can be used to evaluate integrals  $\int_0^5 \sin(x)dx$  and  $\int_3^7 \sin(x)dx$ .

```

int main(){
    integral di(0, 5, sin);              // sin from <math.h>
    double result = di.trapezoidal(100); // call a fcn member
    cout << "Integral from " <<di.lowbd() <<" to " << di.upbd()
        << " is approximately = " << result << '\n';

    di.changebd(3, 7);                  // change bounds
    result = di.trapezoidal(100);
    cout << "Integral from " <<di.lowbd() <<" to " << di.upbd()
        << " is approximately = " << result << '\n';

    result = simpson(di,200);            // invoke a friend
}

```

More complicated definite integrals can be approximated similarly. Efficient integration techniques are discussed in §7.7.

Protecting private members in a class leads to a safer user-defined type that may also be easier to maintain. If the representation of the *integral* class is changed to:

```

class integral {
    double bound[2];                // lower, upper bounds

```

```

    typedef double (*pfn)(double);    // a member that is a type
    pfn integrand;                    // integrand function
    // .... other members
};

```

then only the member functions and friends have to be changed and the user code as in the *main()* function above does not have to be changed. This slight modification of the representation for *integral* has hardly any advantage, but there are many situations in which a change of class representation can improve efficiency or portability. For example, a matrix class can be changed to be represented by a *valarray* (see §7.5) to make use of the high performance computing library.

## 5.2 Copy Constructors and Copy Assignments

By default, a memberwise copy is provided for copy initialization and copy assignment for a class, for which a copy constructor and assignment operator are not explicitly defined such as classes *pt2d* and *integral* in §5.1. This kind of compiler-generated copy constructor and copy assignment should be used when a class does not have a destructor that deallocates space using *delete*, since they may be more efficient than user-defined copy operations. For example,

```

    pt2d d = b;                        // initialize d to b
    c = d;                             // assign d to c

```

where every member of *b* is copied to *d* in the copy initialization in the first statement, and every member of *d* is copied to *c* in the assignment in the second statement. Such default memberwise initialization and assignment were already mentioned in §3.4 for structures.

If the construction of an object of a class has allocated space using the *new* operator, then a destructor is needed to clean up the object when it goes out of scope, using the operator *delete*. The name of the destructor for a class *X* must be  $\sim X()$ . For example,

```

class triangle {
    pt2d* vertices;    // pt2d defined in previous section
public:
    triangle(pt2d, pt2d, pt2d);    // constructor
    ~triangle() { delete[] vertices; } // destructor
    double area() const;           // a function member
};

triangle::triangle(pt2d v0, pt2d v1, pt2d v2) {
    vertices = new pt2d [3]; // 3 vertices in a triangle
    vertices[0] = v0;

```

```

    vertices[1] = v1;
    vertices[2] = v2;
}

void f() {
    pt2d x(1.0,2.0);
    pt2d y(3.0);
    pt2d z;
    pt2d z2(7.0);
    triangle t1(x,y,z);
    triangle t2(x,y,z2);
}

```

When  $f()$  is called, the triangle constructor is called for  $t1$  and  $t2$  to allocate space for their vertex points. At the return of  $f()$ , the space for the vertices are deallocated by the destructor automatically. A destructor need not be called explicitly by the user. Instead, the system calls the destructor implicitly when an object goes out of scope. Note that  $t1$  is constructed before  $t2$ , and  $t2$  is destroyed before  $t1$ . That is, objects are destroyed in the reverse order of construction.

Since *triangle* has a destructor that deallocates space using *delete*, the default memberwise copy constructor and assignment operator would not work correctly in this case:

```

void g(pt2d x, pt2d y, pt2d z) {
    triangle t1(x,y,z);
    triangle t2 = t1;           // copy initialization, trouble
    triangle t3(x,y,z);
    t3 = t1;                   // copy assignment, trouble
}

```

In this code segment, the copy initialization made the member *vertices* of  $t2$  equal to that of  $t1$  (since they are pointers, being equal implies that they point to the same object), and the copy assignment made the member *vertices* of  $t3$  equal to that of  $t1$ . Consequently, the member *vertices* of  $t1$ ,  $t2$ , and  $t3$  point to the same object. At the time  $t1$ ,  $t2$ , and  $t3$  are destroyed when  $g()$  is returned, the same pointer is deleted three times (its behavior is undefined) and results in an error. In general, for a class with a destructor that deletes space allocated by its constructor, the user must define a copy constructor and a copy assignment. For a class  $T$ , they have the form

```

T::T(const T &);           // copy constructor
T& T::operator=(const T&); // copy assignment

```

where *operator* is a keyword that allows the assignment operator  $=$  to be overloaded. In the case of the *triangle* class, they can be defined as

```

triangle::triangle(const triangle & t) { // copy constructor

```

```

    vertices = new pt2d [3];
    for (int i = 0; i < 3; i++) vertices[i] = t.vertices[i];
}

triangle& triangle::operator=(const triangle& t) {
    if (this != &t)    // beware of self-assignment like t = t;
        for (int i = 0; i < 3; i++) vertices[i] = t.vertices[i];
    return *this;
}

```

These two member functions must also be declared in the class *triangle*. Since *this* is a constant pointer that points to the object for which a member function is invoked, *\*this* is then the object. The user-defined copy assignment returns a reference to the object. Notice that new space is allocated in the copy initialization and no new objects are created in the copy assignment, which clearly reflects the fact that initialization and assignment are two distinct operations. With the user-provided copy constructor and assignment above, the function *g()* above should work correctly now. In particular, at the time *t1*, *t2*, and *t3* are destroyed, the three pointers, for which spaces are allocated when they are constructed, are deleted. The redefinition of the assignment operator is called *operator overloading* for *=*. Operator overloading is discussed in more detail in the next chapter.

As another example, a class called *triple* (consisting of three numbers) can be declared as

```

class triple {                                // a triple of numbers
    float* data;
public:
    triple(float a, float b, float c);    // constructor
    ~triple() { delete[] data; }          // destructor
    triple(const triple& t);               // copy constructor
    triple& operator=(const triple& t);    // copy assignment

    friend triple add(const triple&, const triple&); // friend
    void print() const {                  // print out members
        cout << data[0] << " " << data[1] << " "
             << data[2] << '\n';
    }
};

```

Member definitions inside a class declaration are *inline* by default. Other members and friend (also made *inline* since they are short) can be defined as follows.

```

inline triple::triple(float a, float b, float c) {
    data = new float [3];
    data[0] = a; data[1] = b; data[2] = c;
}

```

```

}

inline triple::triple(const triple& t) { // copy constructor
    data = new float [3];
    for (int i = 0; i < 3; i++) data[i] = t.data[i];
}

inline triple& triple::operator=(const triple& t) {
    if (this != &t) // beware of self assignment
        for (int i = 0; i < 3; i++) data[i] = t.data[i];
    return *this;
}

inline triple add(const triple& t, const triple& s) {
    return triple(t.data[0] + s.data[0], t.data[1] + s.data[1],
                 t.data[2] + s.data[2]);
}

```

Now the class *triple* may be used as

```

int main() {
    triple aaa(5,6,7);           // create an object
    triple bbb(10,20,30);
    triple ccc = aaa;             // copy construction
    triple ddd = add(aaa,bbb);    // copy construction

    ccc = add(aaa,ddd);           // copy assignment
    ccc = ccc;                    // self-copying is allowed
    ccc.print();                  // print out members of ccc
}

```

If a class *duple* (consisting of  $n$  numbers) is to be defined, the copy assignment operation may need to take care of assignments of duples with different sizes. To be more precise, consider

```

class duple {                      // a duple of 'size' numbers
    int size;                       // how many numbers?
    double* data;                   // store the numbers
public:
    duple(int n, double* a) {       // constructor
        data = new double [size = n];
        for (int i = 0; i < size; i++) data[i] = a[i];
    }
    ~duple() { delete[] data; }      // destructor
    duple(const duple&);              // copy constructor
    duple& operator=(const duple&);   // copy assignment
}

```

```

    friend duple add(const duple& t, const duple& s);
    void print() const;
};

duple& duple::operator=(const duple& t) { // copy assignment
    if (this != &t) {
        if (size != t.size) { // if sizes are different
            delete[] data; // delete old data
            data = new double [size = t.size]; //space for new data
        }
        for (int i = 0; i < size; i++) data[i] = t.data[i];
    }
    return *this;
}

```

When a duple  $t$  of one size is to be assigned to a duple  $s$  of another size, the space for  $s$  is first deleted and then allocated with the same number of elements in  $t$ . This will make the following code work correctly:

```

double xx[] = { 5, 6, 7};
double* zz = new double [10];
for (int i = 0; i < 10; i++) zz[i] = i + 1;

duple d1(3, xx); // d1 contains 3 numbers
duple d2(8, zz); // d2 contains 8 numbers
d1 = d2; // d1 is resized to have size 8

```

In the assignment  $d1 = d2$ , the object  $d1$  is resized to have the same number of elements as  $d2$ , according to the definition of the copy assignment operator  $=$ . Exercise 5.10.5 asks for the definitions of other members in *duple*.

### 5.3 Friends

A friend can belong to many classes and have access to private members of objects of these classes, while a function, data, or other member can only belong to one class. A friend declaration can be put in either the private or public part of a class; it does not matter where. For example, a matrix-vector multiply function can be declared to be a friend of classes *Mat* and *Vec*:

```

class Mat; // forward declaration
class Vec {
    // .... in addition to other members
    friend Vec multiply(const Mat&, const Vec&);
};

```



```
class Mat {
    // .... in addition to other members
    friend Vec multiply(const Mat&, const Vec&);
};
```

Matrix and vector classes are discussed in more detail in Chapters 6 and 7.

A friend is not a class member and thus does not have a *this* pointer. A member function of one class can be a friend of another class. A class can be a friend of another; in this case all functions of the class become friends. For example,

```
class X {
    void f();
    int g(int);
};
class Y {
    // .... in addition to other members
    friend void X::f();          // f() of X becomes friend of Y
};
class Z {
    // .... in addition to other members
    friend class X;              // all functions of X become
};                               // friends of class Z
```

## 5.4 Static Members

or 单件

Suppose we want to define a class for vectors with default (unit, zero, or other) vectors of certain sizes. One way to do this is to use global variables to store the default vectors. However, too many global variables will make code unmanageable except to its original programmer. Global variables should be kept to the minimum. C++ provides static members for this purpose without introducing global variables or functions. A member that is part of a class but is not part of the objects of the class, is called a *static member*. There is exactly one copy of a static member per class. Note that there is one copy per object for ordinary nonstatic members. Similarly, a function that needs access to members of a class but need not be invoked for a particular object, is called a static member function. For example,

```
class Vec {
private:
    int size;
    double* entry;
    static Vec defaultVec;          // holding default vector
public:
```

```

    Vec(int sz = 0, double* et = 0); // constructor
    Vec(const Vec&);                  // copy constructor
    Vec& operator=(const Vec&);      // copy assignment
    static void setDefault(int, double*);
                                   // function for setting default vector
};

Vec::Vec(int sz, double* et) { // if sz = 0 use default size
    size = sz ? sz : defaultVec.size;
    entry = new double [size];
    if (et == 0) { // if et is null pointer, use default
        for (int i = 0; i < size; i++)
            entry[i] = defaultVec.entry[i];
    } else {
        for (int i = 0; i < size; i++) entry[i] = et[i];
    }
}

```

In the constructor, the default vector size and entries are used when it is not supplied with any arguments. The copy constructor and assignment can be defined accordingly. Static data and function members such as *Vec::defaultVec* and *Vec::setDefault()* must be defined somewhere in the program. Since static members belong to the class, are shared by all objects of the class, and do not belong to a particular object, they are qualified by the name of the class using the scope resolution operator. For example,

```

double ad[] = {5, 6, 7, 8, 9};
Vec Vec::defaultVec(5, ad);
    // defaultVec initialized by ad of size 5

void Vec::setDefault(int sz, double* p) {
    Vec::defaultVec = Vec(sz, p); // reset default vector
}
    // note class qualification Vec::defaultVec

```

Note that a static data member such as *Vec::defaultVec* is defined in the same way as a variable is declared and initialized, very much like the way a function is defined. Although it is private, it may not have to be defined through a member function. A nonstatic data member can not be defined this way. Still, private static members can not be accessed publicly. The default vector can be changed when appropriate. For example,

```

void f() {
    Vec a; // default vec of size 5, a = {5,6,7,8,9}
    Vec b(3); // default vector of size 3, b = {5,6,7}

    double* ey = new double [8];
    for (int i = 0; i < 8; i++) ey[i] = 1;
}

```

```

Vec::setDefault(8, ey);      // reset defaultVec to unit

Vec a2;                      // new default vec of size 8, a2={1,...,1}
Vec b2(3);                   // new default vec of size 3, b2={1,1,1}

double* enb = new double [9];
for (int i = 0; i < 9; i++ ) enb[i] = 8*i*i + 100;
Vec c(9,enb);               // c is a vector constructed from enb.
}

```

Since *a* and *a2* are not supplied with arguments, default vectors (with default sizes and default entries) are constructed. Note the default vector is reset by calling the static member *Vec::setDefault()* before the declaration of *a2*. Thus *a* and *a2* are constructed differently. Similarly, *b* and *b2* are constructed differently although their declarations look exactly the same. In contrast, *c* is constructed according to the arguments supplied to the constructor. From this example, it can be seen that it would be a waste of space if each object such as *b* and *c* had a copy of static member *Vec::defaultVec*. That is, although class *Vec* has member *defaultVec*, an object of *Vec* does not have its own copy of this member. Instead, all objects of *Vec* share this member. In other words, the member *defaultVec* belongs only to class *Vec*. In contrast, every object of *Vec* has a copy of nonstatic members such as *size* and *entry*.

异变的

## 5.5 Constant and Mutable Members

Constant member functions are not supposed to modify objects of a class. However, if a data member is declared to be *mutable*, then it can be changed by a *const* member function. For example, the definite integral class in §5.1 can be redefined so that it has a data member for storing an approximate value of the integral by the Trapezoidal Rule:

```

class integral {
private:
    double lower;
    double upper;
    double (*integrand)(double);

    mutable double value;      // store value from trapezoidal
    mutable bool value_valid;  // value valid or not
public:
    integral(double a, double b, double (*f)(double));
    void changebd(double, double);
    double trapezoidal(int = 100) const;    // const member
};

```

From the user's point of view, the function *trapezoidal()* does not change the state of an object of *integral*. This is why it is declared to be a *const* function. The reason for declaring a *mutable* data member *value* for storing an approximate value of the integral is that *trapezoidal()* might be an expensive process and repeated invocation of it can just return the definite integral value without repetitively calculating it. Now *trapezoidal()* and other members of *integral* can be defined as

```
inline integral::integral(double a, double b,
                        double (*f)(double)) {
    lower = a; upper = b; integrand = f; value_valid = false;
}

inline void integral::changebd(double a, double b) {
    lower = a; upper = b; value_valid = false;
}

double integral::trapezoidal(int n) const { // const member
    if (value_valid == false || n != 100) {
        double h = (upper - lower)/n; // compute integral
        double sum = integrand(lower)*0.5;
        for (int i = 1; i < n; i++) sum += integrand(lower+i*h);
        sum += integrand(upper)*0.5;

        value = sum*h; // update mutable member
        value_valid = true; // update mutable member
    }
    return value;
}
```

This newly defined class *integral* may be used as

```
integral di(7.7, 9.9, log);
double d = di.trapezoidal(); // do the calculation
double e = di.trapezoidal(); // just returns value
```

Observe that the *mutable* member *value* is modified by a *const* member function *trapezoidal()*. In the first call to *trapezoidal()*, the approximate value of the integral is computed and assigned to variable *d*, while in the second call this value is just returned to variable *e* without recomputing it. The computation is not repeated in the second call since *value\_valid* is set to *true* in the first call. This technique is especially useful for adaptive integration methods in which the number of subintervals *n* is not specified as an argument to *trapezoidal()*, but rather is computed by an adaptive algorithm. Since such adaptive algorithms are often expensive, avoiding repetitive execution of them can potentially save a lot of time.

A *mutable* member of a *const* object can also be modified. For example,

```

integral di2(1.1, 2, log);          // log() in <math.h>
const integral di3(1.1, 5, exp);    // a const object
double dd = di2.trapezoidal();      // const member modifies di2
double ee = di3.trapezoidal();      // OK, although di3 is const

di2.changebd(5, 9);                 // OK
di3.changebd(5, 9);                 // error, const object di3

```

The *mutable* member *value* of the *const* object *di3* is modified by the *const* member function *trapezoidal()*. Regular (non-mutable) members of a *const* object can not be modified. Thus *di3.changebd()* is an illegal call since *changebd()* modifies regular members and *di3* is a *const* object.

An example of a class with a *const* data member is given at the end of §5.6. A class with a *static const* data member is presented in §9.1.

## 5.6 Class Objects as Members

Objects of one class can be members of another class. The initialization of such members can be done differently from other members. For example, define a class for line segments (in two dimensions) with a direction that contains members of another class:

```

class line {
    pt2d oneend;          // one end of the line segment
    pt2d otherend;        // the other end of the line segment
    bool direction;       // = 1 if from oneend to otherend
public:                  // = 0 otherwise
    line(pt2d, pt2d, bool); // constructor
    line(pt2d, bool);       // constructor
};

line::line(pt2d a, pt2d b, bool dir): otherend(b), oneend(a) {
    direction = dir;
}

line::line(pt2d b, bool dir): oneend(), otherend(b) {
    direction = dir;
}

```

The first constructor takes two points and a *bool* as arguments. Initialization for class object members (endpoints of the line segment) is done in a member initializer list in the definition of the constructor. The member initializers are preceded by a colon and different member initializers are separated by commas. Initialization for ordinary members (such as *direction* of *line*) is done inside the braces, which can also be done in the initializer

list. The first constructor can create an object of class *line* with *oneend* equal to class object *a*, *otherend* equal to *b*, and *direction* equal to *dir*.

Constructors of the members are called before the body of the containing class' own constructor is executed. The members' constructors are called in the order in which they are declared in the class rather than in the order in which they appear in the initializer list. The members' destructors are called in the reverse order of construction. For example, the construction for member *oneend* as an object of class *pt2d* is done before *otherend* when constructing an object of class *line*.

In the definition of the second constructor above, the endpoint *oneend* of *line* is taken to be the default point (the origin by the default constructor of class *pt2d*; see §5.1). Thus no argument is specified for it. In this case, it can be equivalently written as one of the following.

```
line::line(pt2d b, bool dir): otherend(b) {
    direction = dir;
}
```

```
line::line(pt2d b, bool dir): otherend(b), direction(dir) { }
```

There is an alternative way of defining constructors for some classes. For example, the first constructor can also be defined as

```
line::line(pt2d a, pt2d b, bool dir): otherend(b) {
    oneend = a;
    direction = dir;
}
```

Here *otherend* is initialized to *b*, but *oneend* is first initialized to the origin (default construction for *pt2d*) and then a copy of *a* is assigned to it. Thus there is an efficiency advantage to using the initializer syntax, which avoids default construction and initializes *oneend* to *a* directly. It can even be less efficiently defined as

```
line::line(pt2d a, pt2d b, bool dir) { // less efficient
    oneend = a;
    otherend = b;
    direction = dir;
}
```

These three versions are equivalent except for efficiency.

However, member initializers are essential for member objects of classes without default constructors, for *const* members, and for reference members. For example,

```
class AA {
    const int i;
    line n;
```

```

    line& rn;
    AA(int j,pt2d p,bool d,line& c): i(j), n(p,d), rn(c) { }
};

```

Since *i* is a *const* member, *n* has a type of class *line* that does not have a default constructor, and *rn* is a reference member, they must be initialized in the initializer list, instead of inside the braces. Since constants and references must be initialized, a class with a *const* or reference member can not be default-constructed:

```

struct BB {
    const int i;
    const float f;
    double& d;
};
BB b;          // error, no default constructor is provided

struct CC {
    int i;
    float f;
    double d;
};
CC c;          // OK, use compiler-generated default constructor

```

Objects of classes without constructors or destructors can also be members of a union. A named union is defined as a *struct*, where every member has the same address. A union can have member functions but not static members. In general, a compiler does not know which member of a union is used. Thus a union may not have members with constructors or destructors.

## 5.7 Array of Classes

If a class does not have a constructor or has a default constructor, then arrays of that class can be defined. For example,

```
pt2d ap[100];
```

defines an array of 100 two-dimensional points. Each of the points has been initialized to the origin according to the default constructor of class *pt2d* (see page 176).

However, if a class has constructors that do not provide default values, an array of objects of such a class can not be declared. For example, if a vector class is defined as

```

class Vec {
    double* en;
    int size;

```

```

public:
    Vec(int, double* = 0);      // constructor
    void f(int);                // a test function
};

Vec::Vec(int s, double* d) {
    en = new double [size = s];
    if (d) {                    // if d is not the null pointer
        for (int i = 0; i < size; i++) en[i] = d[i];
    } else {                    // otherwise, use default
        for (int i = 0; i < size; i++) en[i] = 0;
    }
}

```

then an  $n$  by  $n$  square matrix can not be defined directly as an array of  $n$  vectors, each of which represents a row of  $n$  elements:

```

int n = 100;
Vec matrix[n];      // error, Vec has no default constructor

```

where each  $matrix[i]$ ,  $i = 0, 1, \dots, n-1$ , is intended to be a *Vec*, but their sizes can not be specified.

Instead, an array of  $n$  pointers (a double pointer), each of which points to an object of class *Vec*, can be declared. For example,

```

void Vec::f(int n) {
    Vec** tm = new Vec* [n];      // allocate space for matrix
    for (int i = 0; i < n; i++) {  // tm[i] points to an
        tm[i] = new Vec(n);       // object of zero-vector of
    }                             // size n, represent row i

    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) // assign values to entries
            tm[i]->en[j] = 1.0/(i + j + 1);

    for (int i = 0; i < n; i++) delete tm[i];
    delete[] tm;                  // free space after use
}

```

Thus  $tm$  is an array of  $n$  pointers, each of which points to an object of *Vec* that contains  $n$  elements representing a row of a matrix. Since  $tm[i]$  is a pointer, its member should be accessed by using the operator  $\rightarrow$  (in a member function or friend, since  $en$  is a private member of *Vec*). For example,  $tm[i]\rightarrow en[j]$  represents the entry at row  $i$  and column  $j$  of the matrix. This kind of awkward notation can be replaced by using  $tm[i][j]$  through operator overloading; see Chapter 6. Because  $tm[i]$  is a single pointer that points to only one object (although this object contains



$n$  elements in a row), a simple *delete*, instead of *delete[]*, should be used when deallocating space for it. This technique can be very useful and may also be used to define a triangular matrix as an array of pointers; see §11.3 for an application.

## 5.8 Pointers to Members

It is sometimes useful to deal with pointers to member functions of a class. A pointer to a member can be thought of as a position of the member in an object of the class, like an index in an array. It is different from pointers to ordinary functions. Here is a simple example:

```
class X {
    double g(double a) { return a*a + 5.0; }
    double h(double a) { return a - 13; }
public:
    void test(X*, X);
};

typedef double (X::*pf)(double);    // define member pointer

void X::test(X* p, X q) {           // p is pointer, q has type X
    double g5 = p->g(5);             // call member directly
    double h5 = p->h(5);             // call member directly
    double g10 = q.g(10);            // call member directly
    double h10 = q.h(10);            // call member directly

    pf m1 = &X::g;                  // m1 is a pointer to member
    pf m2 = &X::h;                  // m2 is a pointer to member
    double g6 = (p->*m1)(6);          // call thru pointer to member
    double h6 = (p->*m2)(6);          // call thru pointer to member
    double g12 = (q.*m1)(12);        // call thru pointer to member
    double h12 = (q.*m2)(12);        // call thru pointer to member
}
```

Here *pf* is defined to be a type of a pointer to a member of class *X*. A pointer to a member of a class can be obtained by applying the *address-of* operator *&* to a qualified class member name, as in *&X::g* and *&X::h*. The operators *->\** and *.\** can be used to bind pointers to members to class objects. If *m* is a pointer to a member, then *p->\*m* binds *m* to the object pointed to by *p*, and *b.\*m* binds *m* to the object *b*. This technique can be used to evaluate integrals with different integrand functions. Pointers to members can be used in object-oriented finite element analysis code where different coefficient functions and basis functions have to be evaluated.

A static member of a class is not associated with a particular object. Thus a pointer to a static member is just a pointer to an ordinary function. For example,

```
struct Y {
    static void f(int);
};

void (*p)(int) = &Y::f;
           // OK, p is a pointer to an ordinary function
void (Y::*pm)(int) = &Y::f;
           // illegal, ordinary pointer assigned to pm
```

Here  $p$  is declared to be a pointer to an ordinary function that takes an *int* as argument and returns nothing, and  $pm$  is a pointer to a member function of class  $Y$  that takes an *int* and returns nothing. The first initialization is legal since the address of static member  $Y::f$  can be assigned to an ordinary function pointer  $p$ , while the second is an error since the address of a static member is assigned to a class member pointer  $pm$ .

## 5.9 Numeric Methods for Ordinary Differential Equations

Many physical problems are modeled by differential equations. In this section, some basic numeric methods for solving ordinary differential equations are presented.

A first-order ordinary differential equation with a given initial value is usually written in the form

$$\frac{dx}{dt} = f(t, x), \quad t_0 < t \leq T, \quad (5.1)$$

$$x(t_0) = x_0, \quad (5.2)$$

where  $x(t)$  is the unknown function depending on  $t$  (often representing time) and  $f(t, x)$  is a given (source) function depending on  $t$  and  $x$ . The independent variable  $t$  changes from  $t_0$  to  $T$  and the initial value of  $x$  at  $t = t_0$  is given as  $x_0$ . An example is:

$$\frac{dx}{dt} = \frac{(1 - e^t)x}{1 + e^t}, \quad 0 < t \leq 2, \quad (5.3)$$

$$x(0) = 3, \quad (5.4)$$

The exact solution  $x(t)$  can not be found in general. Instead, approximations to the solution  $x(t)$  are sought at certain points  $t_0 < t_1 < \dots < t_N = T$  for a positive integer  $N$ . Let the approximate solution at  $t = t_k$  be

denoted by  $x_k, k = 0, 1, \dots, N$ , with  $x_0$  being the given exact initial value. The objective of a numeric method is to find values  $x_1, x_2, \dots, x_N$ , which approximate the exact solution values  $x(t_1), x(t_2), \dots$ , and  $x(t_N)$ , respectively. One simple such method is to approximate the derivative  $dx/dt$  at  $t = t_k$  by a finite difference  $(x_{k+1} - x_k)/h_k$ , where  $h_k = t_{k+1} - t_k$  is called the grid size. This is because a derivative is defined to be the limit of difference quotients:

$$\left. \frac{dx}{dt} \right|_{t=t_k} = \lim_{h_k \rightarrow 0} \frac{x(t_{k+1}) - x(t_k)}{h_k}.$$

The smaller the grid size  $h_k$  is, the more accuracy may be expected of this approximation in the absence of roundoff errors and numeric instability. That is, from (5.1),

$$\frac{x_{k+1} - x_k}{h_k} \approx \left. \frac{dx}{dt} \right|_{t=t_k} = f(t_k, x(t_k)) \approx f(t_k, x_k),$$

which leads to the so-called *explicit Euler's method*:

$$x_{k+1} = x_k + h_k f(t_k, x_k). \quad (5.5)$$

Since  $x_0$  is given,  $x_1$  can be obtained directly from the formula with  $k = 0$ . Then  $x_2$  can be solved by letting  $k = 1$  and using  $x_1$ . This procedure can go on until  $x_N$  is obtained. For simplicity, equally spaced points are often used; that is,  $t_k = t_0 + kh$ ,  $k = 0, 1, \dots, N$ , with grid size  $h = (T - t_0)/N$ .

Euler's method (5.5) may be programmed as

```
class ode {
    double tini;                // initial time
    double ison;                // initial solution
    double tend;               // end time
    double (*sfn)(double t, double x); // source function
public:
    ode(double t0, double x0, double T,
        double (*f)(double, double)) { // constructor
        tini = t0; ison = x0; tend = T; sfn = f;
    }
    double* euler(int n) const; // Euler with n subintervals
};

double* ode::euler(int n) const { // explicit Euler
    double* x = new double [n + 1]; // approximate solution
    double h = (tend - tini)/n; // grid size
    x[0] = ison; // initial solution
    for (int k = 0; k < n; k++)
        x[k+1] = x[k] + h*sfn(tini + k*h, x[k]);
}
```

```
    return x;
}
```

Here data (*tini* for initial time, *ison* for initial solution, *tend* for ending simulation time, and *sfn* for the source function) about the initial value problem and method *euler()* are encapsulated into a class called *ode*. The private data members can only be accessed by function member *euler()* and others defined later. Copy construction and assignment and a destructor need not be defined for such a class. The user needs to supply the number of subintervals  $N$  to the function *euler()* that returns a pointer to the approximate solution values  $x_0, x_1, \dots, x_N$ .

For the example (5.3)-(5.4), the exact solution happens to be found as  $x(t) = 12e^t/(1 + e^t)^2$ . It can be used to check the accuracy of the approximate solution by Euler's method:

```
double f(double t, double x) {           // source function
    return x*(1 - exp(t))/(1 + exp(t));
}

double exact(double t) {                  // exact solution
    return 12*exp(t)/pow(1 + exp(t), 2);
}

int main() {
    ode exmp(0, 3, 2, f);                 // x(0) = 3, T = 2
    double* soln = exmp.euler(100);       // 100 subintervals

    double norm = 0;
    double h = 2.0/100;                   // grid size
    for (int k = 1; k <= 100; k++)        // compute error
        norm = max(norm, fabs(exact(k*h) - soln[k]));
    cout << "max norm of error by euler's method = "
         << norm << '\n';
}
```

The maximum norm of the error between the exact solution and the approximate solution is 0.00918355. With 200 subintervals ( $h = 1/100$ ), the error is 0.0045747. When  $h = 1/200$ , the error is 0.00228309. It can be observed that the error decreases linearly with the grid size  $h$ . That is, when the grid size  $h$  is halved, the error is reduced by a factor of two. Mathematically, it can be shown that the error is proportional to  $h$ . Such a method is said to have *first-order*. For some problems, the proportionality constant is reasonably small and the explicit Euler's method gives good accuracy. However, for some other problems (see Exercise 5.10.17), the proportionality constant is very large, which causes the method to be very inaccurate. In other words, the explicit Euler's method is not very stable.

A modification to the explicit Euler's method is as follows.

$$x_{k+1} = x_k + h_k f(t_{k+1}, x_{k+1}). \quad (5.6)$$

This is called the *implicit Euler's method*. The only difference is that the evaluation of the source function  $f(t, x)$  is now at  $t = t_{k+1}$ . This scheme requires some nonlinear methods such as Newton's algorithm (see §4.7) to solve for  $x_{k+1}$  from (5.6). Its advantage is that it provides an accurate solution for a wider range of problems than the explicit Euler's method. See Exercise 5.10.17. In other words, the implicit Euler's method is more stable than its explicit counterpart, but requires more computational work.

Based on Euler's method, a predictor-corrector scheme can be constructed as

$$\bar{x}_k = x_k + h_k f(t_k, x_k), \quad (5.7)$$

$$x_{k+1} = x_k + h_k f(t_{k+1}, \bar{x}_k), \quad (5.8)$$

in which a prediction  $\bar{x}_k$  is computed in (5.7) based on the explicit Euler's method and a correction  $x_{k+1}$  is obtained in (5.8) based on the implicit Euler's method. Its advantage is to avoid solving a nonlinear equation as opposed to the implicit Euler's method, and to provide more stability than the explicit Euler's method.

Runge-Kutta methods are explicit since they do not require solving nonlinear equations. A second-order *Runge-Kutta method* is:

$$x_{k+1} = x_k + (F_1 + F_2)/2, \quad (5.9)$$

where

$$\begin{aligned} F_1 &= h_k f(t_k, x_k), \\ F_2 &= h_k f(t_{k+1}, x_k + F_1). \end{aligned}$$

A fourth-order *Runge-Kutta method* is:

$$x_{k+1} = x_k + (F_1 + 2F_2 + 2F_3 + F_4)/6, \quad (5.10)$$

where

$$\begin{aligned} F_1 &= h_k f(t_k, x_k), \\ F_2 &= h_k f(t_k + h_k/2, x_k + F_1/2), \\ F_3 &= h_k f(t_k + h_k/2, x_k + F_2/2), \\ F_4 &= h_k f(t_{k+1}, x_k + F_3). \end{aligned}$$

It can be shown that the error between the exact solution and numeric solution from (5.9) is proportional to  $h^2$  (second-order accuracy) while the error for (5.10) is  $h^4$  (fourth-order accuracy) with the proportionality constant depending on high-order derivatives of the exact solution. In contrast,

Euler's methods (5.5), (5.6), and (5.7)-(5.8) have accuracy proportional to  $h$  (first-order accuracy). When the solution to a problem is well behaved (e.g., its high-order derivatives are not very large), high-order methods such as fourth-order Runge–Kutta methods are preferred. For problems with a transit area where the solution changes very abruptly such as Exercise 5.10.17 with  $\lambda = -500$ , the implicit Euler's method may be preferred.

Now several more functions can be added to class *ode* for these methods. For example,

```
class ode {
public:                                // ... in addition to other members
    double* eulerpc(int n) const;     // predictor--correct euler
    double* rk2(int n) const;         // second-order Runge--Kutta
};

double* ode::eulerpc(int n) const {
    double* x = new double [n + 1];
    double h = (tend - tini)/n;
    x[0] = ison;
    for (int k = 0; k < n; k++) {
        x[k+1] = x[k] + h*sfn(tini + k*h, x[k]);           // predictor
        x[k+1] = x[k] + h*sfn(tini + (k+1)*h, x[k+1]);     // corrector
    }
    return x;
}

double* ode::rk2(int n) const {
    double* x = new double [n + 1];
    double h = (tend - tini)/n;
    x[0] = ison;
    for (int k = 0; k < n; k++) {
        double tp = tini + k*h;
        double f = h*sfn(tp, x[k]);
        x[k+1] = x[k] + 0.5*(f + h*sfn(tp + h, x[k] + f));
    }
    return x;
}
```

See [AP98, CK99, KC96] for more details on numeric methods for ordinary differential equations. Notice that a function pointer is passed to the constructor of class *ode*, which may impose a lot of function calling overhead on function evaluations of *sfn()* inside member functions *euler()*, *eulerpc()*, and *rk2()*. In §7.7, efficient techniques for avoiding such function calling overhead are discussed.

## 5.10 Exercises

- 5.10.1. Extend the class *pt2d*, defined in §5.1, to three-dimensional points. The new class should have several constructors including a default constructor, member functions *move()*, *draw()*, and a friend for calculating the distance from a three-dimensional point to the origin.
- 5.10.2. Use the class for definite integrals defined in §5.1 to evaluate the integrals in Exercises 3.14.22 and 3.14.23.
- 5.10.3. Add printing statements in the constructors and destructor of the class *triangle*, declared in §5.2. For example,

```
triangle::triangle(pt2d v0, pt2d v1, pt2d v2) {
    cout << "allocating space in triangle\n";
    vertices = new pt2d [3];
    // ...
}

triangle::~~triangle() {
    cout << "deleting space in triangle\n";
    delete[] vertices;
}
```

Write a main program calling the function *g()*, defined in §5.2, and compare the outputs without and with user-defined copy constructor and assignment. You should see space being allocated the same number of times as being deleted with user-defined copy constructor and assignment. What do you think will happen without them?

- 5.10.4. Define the member function *triangle::area()*, declared in §5.2, to compute the area of a triangle. Note the area of a triangle in a two-dimensional space with vertices *A*, *B*, and *C* is  $\frac{1}{2}|(B.x - A.x)(C.y - A.y) - (B.y - A.y)(C.x - A.x)|$ . Then use the *triangle* class to find the area of the triangle passing through three two-dimensional points *a*(5, 5), *b*(7, 8), and *c*(20, -4).
- 5.10.5. Define all the function and friend members in class *duple*, as discussed in §5.2. When defining *add()* for the addition of two duples, an error message can be first printed out and the program then exited if the two duples have different sizes. Test a few examples on all the functions, especially copy construction and copy assignment, and print out the data to see if they work as expected.
- 5.10.6. Define a vector class that contains a pointer for the entries, an integer for the size of the vector, and one, two, and maximum norm functions. Test your definitions on a few simple vectors.

- 5.10.7. Define a matrix class that contains a double pointer for the entries, two integers for the dimension (numbers of rows and columns) of the matrix, and one, maximum, and the Frobenius norm functions. Test your definitions on a few simple matrices.
- 5.10.8. Define a friend function for both the matrix class and the vector class in Exercises 5.10.6 and 5.10.7 for matrix-vector multiplication. Test your definitions on a few simple matrices and vectors.
- 5.10.9. Define a class for a triangle given three vertices, each of which is represented by an object of another class for three-dimensional points. Using member initializer lists, define two constructors for it, one with three points as arguments for the three vertices and another with two points (the third vertex is defaulted to be the origin). Also define a member function that moves a triangle to a new location and another member function that prints out the coordinates of the three vertices of a triangle. Create a few triangle objects, move them to new locations, and print out the coordinates of the triangles before and after moving them.
- 5.10.10. Implement the root-finding problem as discussed in §4.7 using a class with the bisection method as a member function and Newton's method as a friend. Then apply it to Exercises 4.8.9 and 4.8.10.
- 5.10.11. A numeric quadrature has the form:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} w_i f(x_i),$$

where  $x_i$  are called quadrature points and  $w_i$  are called weights. In a Gauss quadrature,  $x_i$  and  $w_i$  are chosen such that the quadrature is exact if  $f(x)$  is a polynomial of degree  $p$  for  $p$  as big as possible. Such  $x_i$  are called Gauss points and such  $p$  is called the degree of exactness. It can be shown that the degree of exactness of a Gauss quadrature is  $2n - 1$ , where  $n$  is the number of Gauss points. An example of a Gauss quadrature is:

$$\int_{-1}^1 f(x)dx \approx \frac{8}{9}f(0) + \frac{5}{9}f(-\sqrt{\frac{3}{5}}) + \frac{5}{9}f(\sqrt{\frac{3}{5}}).$$

Implement this Gauss quadrature as a class with Gauss points and weights as static members. Check that this quadrature is exact when the integrand  $f(x)$  is a polynomial of degree less than or equal to 5.

- 5.10.12. In Gauss-Lobatto quadratures, the endpoints of the interval are also quadrature points. An example of a Gauss-Lobatto quadrature is:

$$\int_{-1}^1 f(t)dt \approx \frac{1}{6}f(-1) + \frac{5}{6}f(-\sqrt{\frac{1}{5}}) + \frac{5}{6}f(\sqrt{\frac{1}{5}}) + \frac{1}{6}f(1).$$



Write a class for numerically integrating an integral  $\int_a^b f(x)dx$  with  $n$  (equally spaced) subintervals  $[x_{i-1}, x_i]$  ( $i = 1, \dots, n$ ), on each of which the above Gauss–Lobatto quadrature is applied after a linear transformation from  $[x_{i-1}, x_i]$  onto  $[-1, 1]$ . Hint:  $x(t) = (d - c)t/2 + (d + c)/2$  transforms  $[-1, 1]$  onto  $[c, d]$ .

- 5.10.13. Test the class *integral* defined in §5.5 to see if repeated invocations to function *trapezoidal()* with the default number of subintervals really do not cause recomputation of the definite integral. The member *value* of *integral* is defined to be mutable so that it can be modified by *const* function *trapezoidal()*. Alternatively, *value* may also be defined to be a regular (instead of mutable) member and *trapezoidal()* a nonconstant member function. For what kind of classes does the alternative way seem better?
- 5.10.14. Define an  $n$  by  $n$  lower triangular matrix as a double pointer (or say an array of  $n$  pointers) to *Vec*; see §5.7 for the definition of *Vec*. Each of the  $n$  pointers can be allocated space for a *Vec* of certain number of elements representing a row of the matrix. Assign value  $1/(i + j + 1.0)$  to the entry at row  $i$  and column  $j$  for  $i = 0, 1, \dots, n - 1$  and  $j = 0, 1, \dots, i$ . Finally, deallocate space for the matrix.
- 5.10.15. Define a type that is a pointer to a member of class *integral* as discussed in §5.1. Use pointers to members to call function *trapezoidal()* and evaluate a few definite integrals this way.
- 5.10.16. Implement the implicit Euler’s method and the fourth-order Runge–Kutta method for the class *ode*, discussed in §5.9. Test your implementation on the example (5.3)–(5.4).
- 5.10.17. Compare the accuracy of explicit, implicit, and predictor-corrector Euler’s methods, and the second- and fourth-order Runge–Kutta methods for the initial value problem:  $dx/dt = \lambda x$ ,  $0 < t < 2$ ;  $x(0) = 1$ , with different values of  $\lambda = -10, -50, -100, -500$ , and different grid sizes  $h = 1/50, 1/100, 1/200, 1/400$ , respectively. Its exact solution is  $x(t) = e^{\lambda t}$ . The accuracy should decrease when  $\lambda$  gets smaller (negatively).
- 5.10.18. Suppose that Ms. Li deposits \$10000 in an investment account that pays an annual interest rate of 5% compounded continuously. She then withdraws \$1000 from the account each year in a continuous way starting in year 10. Then  $A(t)$ , the amount of money in the account in year  $t$ , satisfies the differential equation

$$\frac{dA}{dt} = \begin{cases} 0.05A, & \text{for } t < 10, \\ 0.05A - 1000, & \text{for } t \geq 10. \end{cases}$$

How long will the money last? That is, at what time  $t$ , will the account balance  $A(t) = 0$ ? (Answer:  $t = 44.79$  years.)

To maintain a certain life standard, she decides to adjust the withdrawal amount to inflation and withdraws  $\$1000e^{0.03(t-10)}$  in year  $t$  for  $t \geq 10$  (assuming an annual inflation rate of 3% compounded continuously); then the differential equation becomes:

$$\frac{dA}{dt} = \begin{cases} 0.05A, & \text{for } t < 10, \\ 0.05A - 1000e^{0.03(t-10)}, & \text{for } t \geq 10. \end{cases}$$

How long will the money last in this case? (Answer: 30.00 years.)

# 6

## Operator Overloading

Suppose that  $v1$ ,  $v2$ , and  $v3$  are three vectors and  $m$  is a matrix. Mathematically we can write

$$\begin{aligned}
 v3 &= v1 + v2; \\
 v2 &= v1 + m * v3;
 \end{aligned}$$

provided the dimensions of the matrix and vectors are compatible. In C++, we can define classes for vectors and matrices and redefine the meanings of  $+$ ,  $*$ , and  $=$  such that vector addition and matrix-vector multiplication can be written exactly in the same way as the mathematical expressions above. This kind of extension of operators such as  $+$ ,  $*$ , and  $=$  from built-in types to user-defined types is called *operator overloading*. Operator overloading enables the C++ code of many mathematical methods to resemble their algorithms, which can make programming in C++ easier and C++ programs more readable. In this chapter, various issues on operator overloading are discussed. Examples are complex numbers, vectors, and matrices, which are building blocks of many scientific programs. Note that standard C++ libraries include complex numbers (§7.4) and vectors (§7.5 and §10.1.1). A simpler and easier-to-understand version is presented here to illustrate how operators are overloaded. A deferred-evaluation technique is also presented to improve the efficiency of overloaded composite operators. In the final section, operator overloading is applied to solve systems of linear equations using the conjugate gradient method.

## 6.1 Complex Numbers

The name of an operator function is the keyword *operator* followed by the operator itself; for example, *operator+*, *operator\**, and *operator=*. In this section, a class called *Cmpx* for complex numbers is defined and some basic operator functions are provided, some of which are class members and friends and others are just ordinary functions. The purpose is only to show the idea of operator overloading and a user should use the standard library `<complex>` (see §7.4) for computations involving complex numbers. It is declared as

```
class Cmpx {                                // class for complex numbers
private:
    double re;                             // real part of complex number
    double im;                             // imaginary part
public:
    Cmpx(double x=0, double y=0) {re=x; im=y;} // constructor
    Cmpx& operator+=(Cmpx);                 // operator +=, z1 += z2
    Cmpx& operator-=(Cmpx);                 // operator -=, z1 -= z2
    friend Cmpx operator*(Cmpx, Cmpx);      // binary *, z = z1 * z2
    friend ostream& operator<<(ostream&, Cmpx); // cout << z;
    friend istream& operator>>(istream&, Cmpx&); // cin >> z;
};

Cmpx operator+(Cmpx);                      // unary +, eg z1 = + z2
Cmpx operator-(Cmpx);                      // unary -, eg z1 = - z2
Cmpx operator+(Cmpx, Cmpx);                // binary +, eg z = z1 + z2
Cmpx operator-(Cmpx, Cmpx);                // binary -, eg z = z1 - z2
```

Note that the operators `+=`, `-=`, `+`, `-`, `*`, `<<`, and `>>` are overloaded, among which `+` and `-` are overloaded as both unary and binary operators. The operators `+` and `-` are declared as ordinary functions since they do not need direct access to private members of class *Cmpx*. The types *ostream* and *istream* are defined in `<iostream>`. The members, friends, and ordinary functions can be defined as (the order in which they are defined does not matter)

```
inline Cmpx operator+(Cmpx z) {            // unary +, eg z1 = + z2
    return z;
}

inline Cmpx operator-(Cmpx z) {            // unary -, eg z1 = - z2
    return 0 - z;                          // make use of binary -
}                                           // 0 converted to Cmpx(0)

inline Cmpx& Cmpx::operator+=(Cmpx z) { // add-assign, y += z
```

```

    re += z.re; im += z.im;    // increment real and imaginary
    return *this;
}

inline Cmpx& Cmpx::operator--(Cmpx z) { // eg y -= z
    re -= z.re; im -= z.im;    // decrement real and imaginary
    return *this;
}

inline Cmpx operator+(Cmpx a, Cmpx b) { // binary +, z=z1+z2
    return a += b;             // make use of +=: a += b; return a;
}

inline Cmpx operator-(Cmpx a, Cmpx b) { // binary -, z=z1-z2
    return a -= b;             // equivalently: a -= b; return a;
}

inline Cmpx operator*(Cmpx a, Cmpx b) { // eg z = z1 * z2
    return Cmpx(a.re*b.re - a.im*b.im, a.re*b.im + a.im*b.re);
}

ostream& operator<<(ostream& s, Cmpx z) { // output
    s << "(" << z.re << ", " << z.im << ")"; // if z=Cmpx(2,5)
    return s;                                // output (2,5)
}

istream& operator>>(istream& s, Cmpx& z) { // pass reference
    s >> z.re >> z.im;    // input two numbers, written to z
    return s;              // if input 3 6, then z = Cmpx(3,6)
}

```

With the definition for *Cmpx* and the overloaded operators, we can now write:

```

int main() {
    Cmpx a(1,1);           // construction of an object
    Cmpx b = a;             // copy initialization
    Cmpx c = a + b;         // addition and copy initialization
    c -= b;                // subtract and assign
    cout << c << '\n';     // output the complex number c
    c = - b;               // unary operator and assignment
    cout << - c + a*b << '\n';
}

```

Note that operators  $+$  and  $-$  do not directly manipulate the representation of an object but rely on operators  $+=$  and  $-=$ . This can be very useful

when dealing with user-defined types with a lot of data such as matrices and vectors. The usual precedence rules still hold for overloaded operators. For example,  $-c + a * b$  means  $-c + (a * b)$ , instead of  $(-c + a) * b$ . A few more remarks are given below.

### 6.1.1 Initialization

The constructor with default arguments

```
class Cmpx {
public:                // ... in addition to other members
    Cmpx(double x = 0, double y = 0) { re =x; im =y; }
};
```

is equivalent to the following three constructors

```
class Cmpx {
public:                // ... in addition to other members
    Cmpx() { re = 0; im = 0; }           // default constructor
    Cmpx(double x) { re = x; im = 0; }
    Cmpx(double x, double y) { re = x; im = y; }
};
```

Thus the statements

```
Cmpx a;
Cmpx b(3);
Cmpx f = 3;
```

construct  $a$  to be the complex number with real and imaginary parts zero and construct  $b$  and  $f$  to be the complex number with real part 3 and imaginary part 0. In the third statement above, 3 is first constructed into  $Cmpx(3) = Cmpx(3,0)$  and then copied to  $f$  by the compiler-generated copy constructor.

### 6.1.2 Default Copy Construction and Assignment

The default compiler-generated copy constructor and assignment work just fine for this simple class and are preferred over user-defined copy constructor and assignment. A default copy constructor and assignment simply copy all members. The user could also define a copy constructor and assignment:

```
class Cmpx {
public:                // ... in addition to other members
    Cmpx(const Cmpx& c) { re = c.re; im = c.im; }
    Cmpx& operator=(const Cmpx& c) {
        re = c.re; im = c.im;
```

```

    return *this;
}
};

```

However, this is unnecessary, error-prone, and usually less efficient than the compiler-generated default copy constructor and assignment.

### 6.1.3 Conversions and Mixed-Mode Operations

If  $a$  and  $b$  are of *Cmpx*, then in  $a = 2 + b$ , the operand 2 is first converted to *Cmpx*(2) by the constructor and then added to  $b$ . To avoid such conversion and increase efficiency, mixed-mode operations may be defined:

```

class Cmpx {
public:                // ... in addition to other members
    Cmpx& operator+=(double a) {
        re += a;      // only real part is changed
        return *this; // imaginary not affected here
    }
};

inline Cmpx operator+(Cmpx a, double b) {
    return a += b;     // use Cmpx::operator+=(double)
}

inline Cmpx operator+(double a, Cmpx b) {
    return b += a;     // use Cmpx::operator+=(double)
}

```

Now adding a double to a *Cmpx* does not touch the imaginary part of the complex number and thus is simpler and more efficient than adding two *Cmpx*.

## 6.2 Operator Functions

The following operators may be overloaded by a user.

+	−	*	/	%	^	&
	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	=
<<	>>	>>=	<<=	==	!=	<=
>=	&&		++	--	->*	,
->	[]	()	new	new[]	delete	delete[]

The following operators can not be overloaded by a user.

```

:: (scope resolution operator)
. (member selection operator)
.* (member selection through pointer to function)

```

A binary operator can be defined by either a nonstatic member function taking one argument or a nonmember function (a friend or an ordinary function) taking two arguments. If both are defined, overloading resolution determines which one to use or if there is an ambiguity. For any binary operator @, *aa@bb* can be interpreted as either *aa.operator@bb* or *operator@(aa,bb)*. The binary operators + and − are defined as nonmember functions for *Cmpx* in §6.1. They could also be alternatively defined as member functions:

```

class Cmpx {
public:                // ... in addition to other members
    Cmpx operator+(Cmpx);        // binary plus as a member
    Cmpx operator-(Cmpx);        // binary minus as a member
};

inline Cmpx Cmpx::operator+(Cmpx bb) {
    return bb += *this;          // no temporary object needed
}

inline Cmpx Cmpx::operator-(Cmpx bb) {
    Cmpx temp = *this;           // a temporary object is used
    return temp -= bb;           // temp -= bb; return temp;
}

```

When they are defined as class members, *aa + bb* means *aa.operator+(bb)* and *aa + bb* is only a shorthand notation for the explicit function call *aa.operator+(bb)*. Thus if *aa* is of type *Cmpx*, *aa + 1* means *aa + Cmpx(1)*, which in turn is *aa.operator+(Cmpx(1))*. But *1 + aa* will not be legal, since it means *1.operator+(aa)*. Here 1 is not an object of class *Cmpx* and the conversion *Cmpx(1)* is not tried to make a *Cmpx* out of 1, since otherwise it would be possible to change the meaning of operations on built-in types such as *1 + 1*. Member functions can be invoked for objects of their class and no user-defined conversions are applied. Similarly, the statement *1 += aa* is illegal. As a general rule, an operator function intended to accept a basic type as its first operand can not be a member function. Thus binary operators are most commonly defined as friends or ordinary functions. On the other hand, operators that require *lvalue* operands such as =, \*=, and ++, are most naturally defined as members.

Note that a temporary object *temp* is first created by copy construction and then returned (after being decremented by *bb*) in the definition of the binary minus operator when defined as a member function, while no temporary object is created in the operator −=. Thus *a −= b* is not only



a shorthand notation, but also more efficient than  $a = a - b$ . Attempting to avoid such a temporary object in defining the member *operator-* may easily result in an error:

```
Cmpx Cmpx::operator-(Cmpx bb) { // WRONG !
    return *this -= bb;        // this changes the object
}

Cmpx a(9, 5.7);
Cmpx b(3);
Cmpx c = a - b;                // a also equals Cmpx(6, 5.7)
```

This definition not only assigns  $a - b$  to  $c$  but also changes the object  $a$  to  $a - b$  in a statement  $c = a - b$ , since  $a - b$  simply means  $a.operator-(b)$  and  $*this -= bb$  in the definition of  $-$  as a member function above also changes the underlying object. A temporary object may need be created even for member operator  $+$  when passing by reference is used for large objects such as vectors and matrices; see Exercise 6.7.2 and §6.3.

A unary operator, whether prefix or postfix, can be defined by either a nonstatic member function taking no arguments or a nonmember function taking one argument. For any prefix unary operator  $@$ ,  $@aa$  can be interpreted as either  $aa.operator@()$  or  $operator@(aa)$ . For any postfix unary operator  $@$ ,  $aa@$  can be interpreted as either  $aa.operator@(int)$  or  $operator@(aa, int)$ , where the *int* argument is used only to indicate that the operator is postfix, distinguishing it from prefix. The unary operators  $+$  and  $-$  are defined as nonmember functions for *Cmpx* in §6.1. They can also be defined alternatively as member functions:

```
class Cmpx {
public:                // ... in addition to other members
    Cmpx operator+();
    Cmpx operator-();
};

inline Cmpx Cmpx::operator+() { // unary + as member, y+=z
    return *this;
}

inline Cmpx Cmpx::operator-() { // unary - as member, y=-z
    return Cmpx(- re, - im);
}
```

Prefix and postfix operators  $++$  are now defined for *Cmpx* and their use are illustrated:

```
class Cmpx {
public:                // ... in addition to other members
```

```

    Cmpx operator++();           // prefix
    Cmpx operator++(int);       // postfix
};

inline Cmpx Cmpx::operator++() { // prefix as in: ++z
    re++; im++;                // increment both real and imaginary
    return *this;
}

inline Cmpx Cmpx::operator++(int i) { // postfix as in: z++
    Cmpx temp = *this;
    re++; im++;
    return temp;
}

Cmpx aa = 5.0;
Cmpx bb = ++aa;           // bb = Cmpx(6,1), aa = Cmpx(6,1)
Cmpx cc = aa++;           // cc = Cmpx(6,1), aa = Cmpx(7,2)

```

A user can also define `++` to increment only the real part of a complex number, which reflects the flexibility of operator overloading on user-defined types. If  $a$  is an integer,  $++a$  means  $a += 1$ , which in turn means  $a = a + 1$ . For user-defined operators, this is not true unless they are defined this way. For example, a compiler will not generate a definition of  $Z :: operator += ()$  from the definitions of  $Z :: operator + ()$  and  $Z :: operator = ()$ .

The operators `=`, `[]`, `()`, and `->` must be defined as nonstatic member functions to ensure that their first operand is an *lvalue*. The subscripting operator `[]` is used in §6.3, while the function call operator `()` is used in §6.3, §7.6, §10.2.1, and §10.2.2. The dereferencing operator `->` can be used to create “smart pointers” and is discussed in [Str97, LL98].

The operators `=` (assignment), `&` (address-of), and `,` (sequencing) have predefined meanings when applied to class objects. They can be made inaccessible to general users by making them private:

```

class X {
private:
    void operator=(const X&);
    void operator&();
    void operator,(const X&);
};

void f(X a, X b) {
    a = b;           // error, operator = is private
    &a;              // error, operator & is private
    a, b;            // error, operator , is private
}

```

This is sometimes necessary to reduce possible errors or to improve efficiency (see §7.5 for an example).

When an operator is overloaded for many operations with the same base type (e.g., + can be overloaded to add two complex numbers, one complex number and one double, etc.) or different types (e.g., + can add complex numbers, vectors, and matrices), overloading resolution (see §3.8.2, §6.4, and §7.2.2) then comes in to determine which one to use or if there are ambiguities. When the definition of an operator can not be found in the scope in which it is used, then the namespaces of its arguments will be looked up (see §4.1.5).

## 6.3 Vectors and Matrices

In this section, a matrix class called *Mtx*, a vector class called *Vtr*, and their associated operations such as matrix-vector multiply are defined. The declaration of *Vtr* is:

```
class Vtr {
    int lenh;                // number of entries
    double* ets;             // entries of the vector
public:
    Vtr(int, double*);       // constructor
    Vtr(int = 0, double d = 0); // all entries equal d
    Vtr(const Vtr&);         // copy constructor
    ~Vtr(){ delete[] ets; }  // destructor

    int size() const { return lenh; } // return length
    Vtr& operator=(const Vtr&);        // overload =
    Vtr& operator+=(const Vtr &);     // v += v2
    Vtr& operator-=(const Vtr &);     // v -= v2
    double maxnorm() const;           // maximum norm
    double twonorm() const;           // 2-norm
    double& operator[](int i) const { return ets[i]; }
                                    // subscript, eg v[3] = 1.2

    friend Vtr operator+(const Vtr&); // unary +, v = + v2
    friend Vtr operator-(const Vtr&); // unary -, v = - v2
    friend Vtr operator+(const Vtr&, const Vtr&);
                                    // binary +, v = v1 + v2
    friend Vtr operator-(const Vtr&, const Vtr&);
                                    // binary -, v = v1 - v2
    friend Vtr operator*(double, const Vtr&);
                                    // vec-scalar multiply
    friend Vtr operator*(const Vtr&, double);
```

```

// vec-scalar multiply
friend Vtr operator/(const Vtr&, double);
// vec-scalar divide
friend Vtr operator*(const Vtr&, const Vtr&);
// vector multiply
friend double dot(const Vtr&, const Vtr&);
// dot (inner) product
friend ostream& operator<<(ostream&, const Vtr&);
// output operator
};

```

For two vectors  $v = [v_0, v_1, \dots, v_{n-1}]$  and  $w = [w_0, w_1, \dots, w_{n-1}]$ , their dot product is a scalar  $v \cdot w = \sum_{i=0}^{n-1} v_i \bar{w}_i$  (here  $\bar{w}_i$  means the complex conjugate of  $w_i$ ) and their vector product is a vector  $v * w = [v_0 w_0, v_1 w_1, \dots, v_{n-1} w_{n-1}]$ .

The default copy constructor and assignment will not work correctly in this case, since space need be allocated using *new* in the construction of an object of class *Vtr* and deallocated in its destructor. The *operator[]* is overloaded and returns a reference here so that the elements of a vector can be referred to more conveniently. In particular, if *v* is of type *Vtr*, then its *i*th element can be referred to as *v[i]*, instead of *v.ets[i]*, and assignment to *v[i]* is also possible:

```

double a = v[i];      // better notation than v.ets[i]
v[i] = 10;            // operator[] returns a reference

```

Here are the definitions of the members and friends of class *Vtr*:

```

inline void error(char* v) {          // auxiliary fcn
    cout << v << ". program exited\n"; // include <iostream>
    exit(1);                          // include <stdlib.h>
}

Vtr::Vtr(int n, double* abd) {        // constructor
    ets = new double [lenth = n];
    for (int i = 0; i < lenth; i++) ets[i] = *(abd + i);
}

Vtr::Vtr(int n, double a) {           // constructor
    ets = new double [lenth = n];
    for (int i = 0; i < lenth; i++) ets[i] = a;
}

Vtr::Vtr(const Vtr & v) {             // copy constructor
    ets = new double [lenth = v.lenth];
    for (int i = 0; i < lenth; i++) ets[i] = v[i];
}

```

```

Vtr& Vtr::operator=(const Vtr& v) { // overload =
    if (this != &v) { // beware of self-assignment
        if (lenth != v.lenth ) error("bad vector sizes");
        for (int i = 0; i < lenth; i++) ets[i] = v[i];
    }
    return *this;
}

Vtr& Vtr::operator+=(const Vtr& v) { // add-assign, u += v
    if (lenth != v.lenth ) error("bad vector sizes");
    for (int i = 0; i < lenth; i++) ets[i] += v[i];
    return *this;
}

Vtr& Vtr::operator-=(const Vtr& v) { // subtract-assign
    if (lenth != v.lenth ) error("bad vector sizes");
    for (int i = 0; i < lenth; i++) ets[i] -= v[i];
    return *this;
}

inline Vtr operator+(const Vtr& v) { // usage: u = + v
    return v; // unary +
}

inline Vtr operator-(const Vtr& v) { // usage: u = - v
    return Vtr(v.lenth) - v; // unary -
}

Vtr operator+(const Vtr& v1, const Vtr& v2) { // addition
    if (v1.lenth != v2.lenth ) error("bad vector sizes");
    Vtr sum = v1; // would cause error without copy constructor
    sum += v2; // eg, v1 would be changed in v =v1 +v2;
    return sum; // since sum.ets would equal v1.ets
}

Vtr operator-(const Vtr& v1, const Vtr& v2) { // subtract
    if (v1.lenth != v2.lenth ) error("bad vector sizes");
    Vtr sum = v1; // create a temporary object
    sum -= v2;
    return sum; // members of sum returned
} // sum is then destroyed

Vtr operator*(double scalar, const Vtr& v) {
    Vtr tm(v.lenth); // scalar-vector multiply

```

```

    for (int i = 0; i < v.lenth; i++) tm[i] = scalar*v[i];
    return tm;
}

inline Vtr operator*(const Vtr& v, double scalar) {
    return scalar*v;           // scalar-vector multiply
}

Vtr operator*(const Vtr& v1, const Vtr& v2) { // multiply
    if (v1.lenth != v2.lenth ) error("bad vector sizes");
    int n = v1.lenth;
    Vtr tm(n);
    for (int i = 0; i < n; i++) tm[i] = v1[i]*v2[i];
    return tm;
}

Vtr operator/(const Vtr & v, double scalar) {
    if (!scalar)           // vector scalar divide
        error("division by zero in vector-scalar division");
    return (1.0/scalar)*v;
}

double Vtr::twonorm() const {           // two (euclidean) norm
    double norm = ets[0]*ets[0];
    for (int i = 1; i < lenth; i++) norm += ets[i]*ets[i];
    return sqrt(norm);
}

double Vtr::maxnorm() const {           // maximum norm
    double norm = fabs(ets[0]);
    for (int i = 1; i < lenth; i++)
        norm = max(norm, fabs(ets[i])); // max() in <algorithm>
    return norm;
}

double dot(const Vtr &v1, const Vtr& v2) { // dot product
    if (v1.lenth != v2.lenth ) error("bad vector sizes");
    double tm = v1[0]*v2[0];
    for (int i = 1; i < v1.lenth; i++) tm += v1[i]*v2[i];
    return tm;
}

ostream& operator<<(ostream& s, const Vtr& v ) { // output
    for (int i =0; i < v.lenth; i++ ) {
        s << v[i] << " ";
    }
}

```

```

    if (i%10 == 9) s << "\n"; // print 10 elmts on a line
}
return s;
}

```

Note that without the user-defined copy constructor, erroneous results would occur in statements such as  $v = v1 + v2$ , which would cause the summation  $v1 + v2$  to be assigned to both  $v1$  and  $v$ . The reason is that in the definition of the binary operator  $+$  above, the variable *sum* is copy-constructed from  $v1$  and thus *sum.ets* would equal *v1.ets* with a compiler-generated copy constructor. Thus any update to *sum* would occur to entries of  $v1$  as well. Compared to class *Cmpx* in §6.1, temporary objects are created for *Vtr* in the definitions of member functions such as the binary operator  $+$ . This may make operation  $v1 += v2$  noticeably more efficient than a seemingly equivalent one:  $v1 = v1 + v2$ . The code is also written such that the program will exit if two vectors of different sizes are accidentally added or subtracted.

Now vector operations in C++ can be written in a similar way to conventional mathematical notation. For example,

```

void f() {
    int m = 500;
    double* v = new double [m];
    for (int i = 0; i < m; i++) v[i] = i*i + 10;
    Vtr v1(m, v);           // create v1 from v
    Vtr v2(m);              // zero vector of size m
    for (int i = 0; i < m; i++) v2[i] = 5*i - 384;
    Vtr v3(m, 5.8);        // every entry equals 5.8

    Vtr v4 = - v1 + 3.3*v3; // resemble mathematics
    v4 += v2;
    cout << v4;            // output vector

    Vtr v5 = - v1*v4;       // vector multiply
    double a = dot(v1, v5); // dot product
}

```

Below is a class *Mtx* for matrices, in which some operations such as matrix addition and matrix-vector multiplication are defined. To compare the difference in defining member functions and friends, matrix addition is implemented as a member function and matrix subtraction as a friend. This implementation detail should not have much effect on how the *Mtx* class is going to be used.

```

class Mtx {
private:
    int nrow;                // number of rows

```

```

    int ncols;                // number of columns
    double** ets;             // entries of matrix
public:
    Mtx(int n, int m, double**); // constructor (n by m)
    Mtx(int n, int m, double d = 0); // all entries equal d
    Mtx(const Mtx &);           // copy constructor
    ~Mtx();                     // destructor

    Mtx& operator=(const Mtx&); // overload =
    Mtx& operator+=(const Mtx&); // overload +=
    Mtx& operator-=(const Mtx&); // overload -=
    Vtr operator*(const Vtr&) const; // matrix vec multiply
    double* operator[](int i) const { return ets[i]; }
        // subscript, entry at row i and column j is [i][j]
    double& operator()(int i, int j) { return ets[i][j]; }
        // subscript, entry at row i and column j is (i,j)

    // Implement plus as a member fcn. minus could also be
    // so implemented. But I choose to implement minus as
    // a friend just to compare the difference
    Mtx& operator+() const; // unary +, m1 = + m2
    Mtx operator+(const Mtx&) const; // binary +, m = m1+m2
    friend Mtx operator-(const Mtx&); // unary -, m1 = - m2
    friend Mtx operator-(const Mtx&, const Mtx&); // binary -
};

```

The definitions of these members and friends can be:

```

Mtx::Mtx(int n, int m, double** dbp) { // constructor
    nrows = n;
    ncols = m;
    ets = new double* [nrows];
    for (int i = 0; i < nrows; i++) {
        ets[i] = new double [ncols];
        for (int j = 0; j < ncols; j++) ets[i][j] = dbp[i][j];
    }
}

Mtx::Mtx(int n, int m, double a) { // constructor
    ets = new double* [nrows = n];
    ncols = m;
    for (int i = 0; i < nrows; i++) {
        ets[i] = new double [ncols];
        for (int j = 0; j < ncols; j++) ets[i][j] = a;
    }
}

```



```

Mtx::Mtx(const Mtx & mat) {           // copy constructor
    ets = new double* [nrows = mat.nrows];
    ncols = mat.ncols;
    for (int i = 0; i < nrows; i++) {
        ets[i] = new double [ncols];
        for (int j = 0; j < ncols; j++) ets[i][j] = mat[i][j];
    }
}

inline Mtx::~~Mtx(){                  // destructor
    for (int i = 0; i < nrows; i++) delete[] ets[i];
    delete[] ets;
}

Mtx& Mtx::operator=(const Mtx& mat) { // copy assignment
    if (this != &mat) {
        if (nrows != mat.nrows || ncols != mat.ncols)
            error("bad matrix sizes");
        for (int i = 0; i < nrows; i++)
            for (int j = 0; j < ncols; j++)
                ets[i][j] = mat[i][j];
    }
    return *this;
}

Mtx& Mtx::operator+=(const Mtx& mat) { // add-assign
    if (nrows != mat.nrows || ncols != mat.ncols)
        error("bad matrix sizes");
    for (int i = 0; i < nrows; i++)
        for (int j = 0; j < ncols; j++)
            ets[i][j] += mat[i][j];
    return *this;
}

Mtx& Mtx::operator-=(const Mtx& mat) { // subtract-assign
    if (nrows != mat.nrows || ncols != mat.ncols)
        error("bad matrix sizes");
    for (int i = 0; i < nrows; i++)
        for (int j = 0; j < ncols; j++)
            ets[i][j] -= mat[i][j];
    return *this;
}

inline Mtx& Mtx::operator+( ) const { // usage: m1 = + m2

```

```

    return *this;
}

inline Mtx operator-(const Mtx & mat) {    // m1 = - m2
    return Mtx(mat.nrows, mat.ncols) - mat;
}

Mtx Mtx::operator+(const Mtx & mat) const { // m = m1 + m2
    if (nrows != mat.nrows || ncols != mat.ncols)
        error("bad matrix sizes");
    Mtx sum = *this;          // user-defined copy constructor
    sum += mat;               // is important here
    return sum;               // otherwise m1 would be changed
}

Mtx operator-(const Mtx& m1, const Mtx& m2) { // m=m1-m2
    if(m1.nrows != m2.nrows || m1.ncols != m2.ncols)
        error("bad matrix sizes");
    Mtx sum = m1;
    sum -= m2;
    return sum;
}

Vtr Mtx::operator*(const Vtr& v) const { // u = m*v
    if (ncols != v.size())
        error("matrix and vector sizes do not match");
    Vtr tm(nrows);
    for (int i = 0; i < nrows; i++)
        for (int j = 0; j < ncols; j++)
            tm[i] += ets[i][j]*v[j];
    return tm;
}

```

Then the following code can be conveniently written.

```

int main() {
    int k = 300;
    double** mt = new double* [k];
    for (int i = 0; i < k; i++) mt[i] = new double [k];
    for (int i = 0; i < k; i++)
        for (int j = 0; j < k; j++) mt[i][j] = 2*i*j + i + 10;

    Mtx m1(k, k, mt);          // construct m1 from mt
    Mtx m2(k, k, 5);           // construct m2, all entries = 5
    Mtx m3(k, k);               // construct m3, all entries = 0
    for (int i = 0; i < k; i++) // update entries of m3

```

```

    for (int j = 0; j < k; j++) m3[i][j] = 1/(2*i + j + 5.7);

    m3 += - m1 + m2;           // resemble mathematics
    m1 -= m3;                  // very readable

    Vtr vv(k);
    for (int i = 0; i < k; i++ ) vv[i] = 5*i + 3;
    vv = m3*vv;                // resemble mathematics
}

```

Since the function call operator `()` is overloaded, a matrix entry can also be accessed using FORTRAN notation such as  $m3(5, 7) = 10$ .

Note that pass by reference has been used when possible in the definition of member functions and friends of classes *Mtx* and *Vtr*. This is more efficient when dealing with large objects than pass by value. In the latter case operands are copied according to the copy constructor. Pass by value is used in *Cmpx*, where only two doubles of a complex number are copied. Since matrices and vectors often have many entries, copying their members would impose significant overhead. For example, if the binary plus operator is defined through pass by value:

```

Mtx Mtx::operator+(Mtx mat) {    // usage: m1 + m2
    Mtx sum = *this;             // user-defined copy constructor
    sum += mat;                  // is important here
    return sum;                  // otherwise m1 would be changed
}

```

then in the statement  $m = m1 + m2$  (or equivalently  $m = m1.operator+(m2)$ ), members of the argument  $m2$  will be copied to the parameter  $mat$  by calling the copy constructor. The return value  $sum$  is also copied to the calling environment (as a temporary object, which is then assigned to variable  $m$  by the copy assignment operator; this temporary object can often be optimized away). Using pass by reference just creates an alias, instead of copying all the entries of a matrix. It would be more efficient if the return value could be passed by reference. However, pass by reference for a local variable (such as  $sum$  in the definition of the binary  $+$  above) can not be used (see §3.8.4).

Pointers can not be used because it is not possible to redefine the meaning of an operator acting on a pointer or any other built-in type.

The vector class *Vtr* (especially its template form in Chapter 7) may be used to replace one-dimensional built-in arrays (see §3.3) since it provides dynamic memory allocation and does not require the user to allocate and deallocate space explicitly. For example,

```

double g(int n) {
    Vtr a(n);                    // allocate dynamic memory
    for (int i = 0; i < n; i++)

```

```

        a[i] = std::sin(i*i);    // can be used like an array
    return a.twonorm();          // space freed automatically
}

```

Besides, it provides basic vector operations and the user can easily extend it to meet specific needs. However, it may not be as efficient as the standard library `<vector>` (see §10.1.1) and `<valarray>` (see §7.5), which also provide many more operations than *Vtr*.

Since C++ standard libraries do not include matrices, the matrix class *Mtx* (especially its template form in Chapter 7) may be used in place of two-dimensional built-in arrays (see §3.3). For example,

```

double h(int n, int m) {
    Mtx mx(n, m);                // allocate dynamic memory
    for (int i = 0; i < n; i++)    // for an n by m array
        for (int j = 0; j < m; j++)
            mx[i][j] = 1.2/(i + j + 1); // can be used like array
    return mx.maxnorm();          // space freed implicitly
}

```

From a user's point of view, no double pointers and memory management need be manipulated directly.

## 6.4 Explicit and Implicit Conversions

A constructor with one argument also defines a conversion. For example,

```

class Cmpx {
public:                // ... in addition to other members
    Cmpx(double a);    // constructor from double to complex
    Cmpx(const Cmpx &); // copy constructor
};

void f(Cmpx);
Cmpx z = 2.2;          // initialize z to Cmpx(2.2)
f(5.5);                // call fcn f with argument Cmpx(5.5)

```

The constructor first implicitly converts 2.2 into *Cmpx*(2.2), and then initializes *z* with the complex number *Cmpx*(2.2). This is called *implicit conversion*. For some types, this conversion may not be desirable. For example, conversion from *int* to an enumeration type may cause truncation. Implicit conversion can be suppressed by declaring a constructor to be *explicit*. With an *explicit* constructor, conversion can not be done implicitly. For example,

```

class Cmpx {
public:                // ... in addition to other members

```

```

    explicit Cmpx(double a); // explicit constructor
    Cmpx(const Cmpx &);      // copy constructor
};

void f(Cmpx);
Cmpx y = 2.2; // error, implicit conversion disallowed
Cmpx z = Cmpx(2); // OK, explicit conversion
f(5.5); // error, can not convert 5.5 to Cmpx(5.5)
f(Cmpx(5.5)); // OK
f(z); // OK

```

Conversion operators can also be defined to convert one type to another. If  $X$  is a class and  $T$  is a type, then the member function  $X :: \text{operator } T()$  defines a conversion from  $X$  to  $T$ . For example, one can define a class for 4-bit nonnegative integers, called *tiny*, that can be assigned to and from integers:

```

class tiny {
    char v;
    void assign (int i) { // for range checking
        if (i & ~0xf) { cout << "range error\n"; exit (1); }
        v = i;
    }
public:
    tiny(int i) { assign(i); } // constructor
    tiny& operator=(int i) { assign(i); return *this; }
    operator int() const { return v; } // conversion operator
};

```

Note that  $0xf$  is a hexadecimal number and  $(f)_{16} = (1111)_2 = (15)_{10}$ , where  $(n)_b$  represents number  $n$  in base  $b$ . The bitwise expression  $i \& \sim 0xf$  yields a value having a bit representation with the four low-order bits zero and all other bits the same as the high-order bits of  $i$ . The hex number  $\sim 0xf$  is called a *mask* for high-order bits beyond the four low-order bits, and  $0xf$  a *mask* for the four low-order bits. Range checking is made when a *tiny* is initialized by an *int* and when an *int* is assigned to a *tiny*. No range checking is needed when copying a *tiny* so that the default copy constructor and assignment work just fine for this simple class. Now implicit conversion from *tiny* to *int* is handled by the conversion operator  $\text{tiny} :: \text{operator int}()$ . Note that the type being returned is part of the name of the conversion operator and can not be repeated in the function:

```

class tiny {
public:
    // ... in addition to other members
    operator int() const { return v; } // correct
};

```

```

class tiny {
public:                // ... in addition to other members
    int operator int() const { return v; } // wrong !!!
};                    // no return type needed

```

Now variables of *tiny* and *int* can be converted to each other freely:

```

tiny t = 12;        // int is assigned to tiny
int i = t;          // tiny is converted to int
cout << int(t);     // convert t to int and print it out
tiny tt = 16;       // range error occurs, program exits.

```

Note that ambiguity can arise with user-defined conversions and user-defined operators. For example,

```

tiny operator+(tiny t, tiny s) { // overload + for tiny
    return tiny(int(t) + int(s));
}

```

```

void f(tiny t, int i) {
    t + i; // error, ambiguous: int(t) + i or t + tiny(i) ?
}

```

The user should replace  $t + i$  by  $\text{int}(t) + i$  or  $t + \text{tiny}(i)$  according to what is intended.

In some cases, a value of the desired type can be constructed by repeated use of constructors or conversion operators. This must be done by explicit conversion. Only one level of user-defined implicit conversion is legal. User-defined conversions are considered only if they are necessary to resolve a call. For example,

```

struct X { X(int); X(double); }; // X has two constructors
struct Y { Y(int); Y(double); }; // Y has two constructors
struct Z { Z(X); };              // Z has one constructor
X f(X);
Y f(Y);
Z g(Z);
X h(X);
X h(double);

```

```

f(5);           // error, ambiguous: f(X(5)) or f(Y(5)) ?
f(X(5));        // OK
f(Y(3.14));     // OK
f(3.14);        // error
g(5.6);         // error, g(Z(X(5.6))) is not tried.
h(5);           // OK, h(double(5)) is used, h(X(5)) not used

```

In the call  $g(5.6)$ , two user-defined conversions  $g(Z(X(5.6)))$  would be needed and thus are not tried. Only one level of user-defined implicit con-

version is legal. In the call  $h(5)$ , the standard conversion  $h(\text{double}(5))$  for the argument from *int* to *double* is preferred over the user-defined conversion  $h(X(5))$ .

## 6.5 Efficiency and Operator Overloading

When dealing intensively with large objects such as matrices and vectors, efficiency may be a predominating factor to be considered in many scientific computing applications. It was mentioned in §6.3 that creating temporary objects or copying large objects could slow down a program.

In addition, multiple loops in composite operations can also affect efficiency. For example, in the so-called vector *saxpy* (mnemonic for scalar  $a$   $x$  plus  $y$ ) operation  $z = a * x + y$  for vectors  $x, y, z$  and scalar  $a$ , one loop is needed for vector-scalar multiplication  $a * x$ , another loop for vector addition  $a * x + y$ , and yet another for copying the result to vector  $z$ , when using the vector class *Vtr* defined in §6.3. Also temporary objects may need be created for holding  $a * x$  and  $a * x + y$  in this process (besides the construction and destruction of temporary objects for automatic variables in the definitions of operators  $*$  and  $+$  for *Vtr*). That is, a few loops are needed for a simple *saxpy* operation. A much more efficient and traditional way for defining the *saxpy* operation is to write only one loop without any temporary objects, as is normally done in FORTRAN 77 and C:

```
inline void saxpy(double a, const Vtr& x, const Vtr& y, Vtr& z){
    for (int i = 0; i < z.size(); i++) z[i] = a*x[i] + y[i];
}
```

But this would sacrifice readability and defeat the purpose of operator overloading. That is, instead of writing

```
z = a*x + y;
```

one would have to write a less expressive statement

```
saxpy(a, x, y, z);
```

in order to improve efficiency.

How can one achieve the same efficiency as in the traditional way while still using operator overloading?

Below is a technique to achieve this. The idea is to defer the evaluations of intermediate suboperations in a composite operation until the end of the operation. First, define a class called *Sax* for scalar-vector multiplication, which does not do the evaluation but simply constructs an object holding the references to the scalar and vector to be multiplied:

```
struct Sax {
    const double& a;    // reference is used to avoid copying
```

```

    const Vtr& x;
    Sax(const double& d, const Vtr& v) : a(d), x(v) { }
};

```

```

inline Sax operator*(const double& d, const Vtr& v) {
    return Sax(d,v);          // overload operator *
}

```

Then define a class called *Saxpy* for vector *saxpy* operations (Again, to avoid creating temporary objects, evaluations of suboperations are deferred).

```

struct Saxpy {
    const double& a;    // reference is used to avoid copying
    const Vtr& x;
    const Vtr& y;
    Saxpy(const Sax& s, const Vtr& u) : a(s.a), x(s.x), y(u) { }
};

```

```

inline Saxpy operator+(const Sax& s, const Vtr& u) { //a*x+y
    return Saxpy(s, u);          // overload +
}

```

```

inline Saxpy operator+(const Vtr& u, const Sax& s) { //x+a*y
    return Saxpy(s, u);          // overload +
}

```

Finally, define a copy constructor and a copy assignment from *Saxpy* to *Vtr*, in which the evaluation of the *saxpy* operation is done:

```

class Vtr {
    int lenth;          // number of entries
    double* ets;        // entries of vector
public:
    Vtr(int, double*);  // constructor
    Vtr(int, double = 0); // constructor
    Vtr(const Vtr&);    // copy constructor
    Vtr(const Saxpy&);  // constructor from Saxpy
    ~Vtr(){ delete[] ets; } // destructor

    Vtr& operator=(const Vtr&);    // overload assignment
    Vtr& operator=(const Saxpy&);  // overload assignment
    int size() const { return lenth; }
    double& operator[](int i) const { return ets[i]; }
};

```

```

Vtr::Vtr(const Saxpy& sp) {

```



```

ets = new double [lenth = sp.x.size()];
for (int i = 0; i < lenth; i++)
    ets[i] = sp.a*sp.x[i] + sp.y[i];
}

```

```

Vtr& Vtr::operator=(const Saxpy& sp) {
    for (int i = 0; i < lenth; i++)
        ets[i] = sp.a*sp.x[i] + sp.y[i];
    return *this;
}

```

With the new definition of class *Vtr*, the usual scalar-vector multiplication operator (see §6.3) should not be defined, since otherwise ambiguity would arise. Notice that *Vtr* is used when defining *Saxpy*, and *Saxpy* is used when defining *Vtr*. A forward declaration is actually needed to resolve this chaining. With a forward declaration for *Vtr*, the name of class *Vtr* can be used in *Sax* and *Saxpy* before *Vtr* is defined, as long as its use does not require the size of *Vtr* or its members to be known.

Now a *saxpy* operation can be evaluated without creating any temporary *Vtr* objects and with only one loop:

```

int main() {
    int k = 50000;
    double* p = new double [k];
    for (int i = 0; i < k; i++ ) p[i] = i;
    Vtr v(k, p);           // create a Vtr object using p
    Vtr w(k, 5);           // create another Vtr object

    Vtr u = 5*v + w;       // efficient saxpy operation
    v = u + 3.14*w;        // efficient saxpy operation
}

```

In fact, a *saxpy* operation  $z = a * x + y$  is now equivalent to

$$z.operator=(Saxpy(Sax(a, x), y)).$$

Due to inlining for operators  $*$  and  $+$  above, this should be as efficient as the traditional *saxpy()* function. This deferred evaluation technique can be generalized into what are called *expression templates*; see §7.7.3.

Run-time comparisons can be made for the three versions of the *saxpy* operation based on the straightforward operator overloading as defined in §6.3, the deferred-evaluation operator overloading as defined in this section, and the traditional procedural (C or FORTRAN) style defined as function *saxpy()* early in this section. For vectors of size 50000, *saxpy* operations are done repeatedly 1000 times. The three versions take 152, 57, and 72 seconds, respectively, on a Pentium PC (100 MHz) running Linux with a GNU C++ compiler. The same operations take 104, 25, and 25 seconds, respectively,

on a SUN Ultra Workstation (167 MHz) running Solaris with a GNU C++ compiler. They take 67, 14, and 19 seconds, respectively, on a Pentium II PC (400 MHz) running Linux with a GNU C++ compiler. On an SGI machine running UNIX, they take 78, 18, and 26 seconds, respectively. On these four different machines, the straightforward operator overloading is three to five times as slow as the deferred-evaluation operator overloading. The latter is even faster than the traditional procedural function call on three of the four machines, which is hard to explain.

## 6.6 Conjugate Gradient Algorithm

In this section, the conjugate gradient algorithm is presented and coded in C++ for solving systems of linear equations. The details of this algorithm are given in Chapter 11.

Let  $\langle \xi, \eta \rangle$  denote the usual dot product in  $\mathcal{R}^n$  and  $A = (a_{i,j})$  a real  $n$  by  $n$  symmetric and positive definite matrix; that is,

$$a_{i,j} = a_{j,i}, \quad \text{for all } i, j = 0, 1, \dots, n-1, \quad (6.1)$$

$$\langle A\xi, \xi \rangle > 0, \quad \text{for all } \xi \in \mathcal{R}^n \text{ and } \xi \neq 0. \quad (6.2)$$

Given an initial guess  $x_0$ , the conjugate gradient algorithm iteratively constructs a sequence  $\{x_k\}$ , which converges to the solution of the matrix equation  $Ax = b$ . It can be stated as follows.

**Algorithm 6.6.1** *Taking an initial guess  $x_0 \in \mathcal{R}^n$  and setting  $r_0 = b - Ax_0$ ,  $p_0 = r_0$ , construct the sequence  $x_k$ , for  $k = 0, 1, 2, \dots$ ,*

$$\alpha_k = \frac{\langle r_k, r_k \rangle}{\langle Ap_k, p_k \rangle}, \quad (6.3)$$

$$x_{k+1} = x_k + \alpha_k p_k, \quad (6.4)$$

$$r_{k+1} = r_k - \alpha_k Ap_k, \quad (6.5)$$

$$\beta_k = \frac{\langle r_{k+1}, r_{k+1} \rangle}{\langle r_k, r_k \rangle}, \quad (6.6)$$

$$p_{k+1} = r_{k+1} + \beta_k p_k. \quad (6.7)$$

Equation (6.4) shows how a more accurate solution  $x_{k+1}$  is obtained from an old iterate  $x_k$ . The vectors  $p_k$  are called search directions and the scalars  $\alpha_k$  search lengths. It can be shown easily that  $r_{k+1} = r_k - \alpha_k Ap_k = b - Ax_{k+1}$ . Thus  $r_{k+1}$  represents the residual corresponding to the approximate solution  $x_{k+1}$ .

This algorithm can be implemented with only one matrix-vector multiplication and two vector dot products, two scalar-vector multiplications, and one vector *saxpy* operation per iteration. The iterative process can be stopped when a norm of the residual  $r_k = b - Ax_k$  is smaller than a

prescribed small number  $\epsilon$  or the total number of iterations has exceeded a given number. The closer is the initial guess  $x_0$  to the exact solution, the faster is the convergence. However, the initial guess  $x_0$  is often taken to the zero vector when not much information is known about the exact solution. It can be shown that, for Hermitian positive definite coefficient matrix  $A$ , the iterates  $x_k$  converge to the exact solution within  $n$  iterations no matter what the initial guess  $x_0$  is, in the absence of roundoff errors (i.e., in exact arithmetics).

Making use of the vector and matrix classes defined in previous sections, it can be coded as

```
class Mtx {
public:          // ... in addition to other members
    int CG(Vcr& x, const Vcr& b, double& eps, int& iter);
/* Conjugate gradient method for Ax = b.
   it returns 0 for successful return and 1 for breakdowns

   A: symmetric positive definite coefficient matrix
   x: on entry: initial guess; on return: approximate soln
   b: right side vector
   eps: on entry:  stopping criterion, epsilon
        on return: absolute residual in two-norm
                for approximate solution
   iter: on entry: max number of iterations allowed;
        on return: actual number of iterations taken.
*/
};

int Mtx::CG(Vcr& x, const Vcr& b, double& eps, int& iter) {
    if (nrows != b.size())
        cout << "matrix and vector sizes do not match\n";
    const int maxiter = iter;
    Vcr r = b - (*this)*x;          // initial residual
    Vcr p = r;                      // p: search direction
    double zr = dot(r,r);           // inner prod of r and r
    const double stp = eps*b.twonorm(); // stopping criterion

    if (r.twonorm() == 0) { // if initial guess is true soln,
        eps = 0.0;         // return. Otherwise division by
        iter = 0;          // zero would occur.
        return 0;
    }

    for (iter = 0; iter < maxiter; iter++) { // main loop
        Vcr mp = (*this)*p;          // matrix-vector multiply
```

```

        double alpha = zr/dot(mp,p);    // divisor=0 only if r=0
        x += alpha*p;                   // update iterative soln
        r -= alpha*mp;                  // update residual
        if (r.twonorm() <= stp) break;  // stop if converged
        double zrold = zr;
        zr = dot(r,r);                  // dot product
        p = r + (zr/zrold)*p;           // zrold=0 only if r=0
    }

    eps = r.twonorm();
    if (iter == maxiter) return 1;
    else return 0;
}                                     // end CG()

```

With operator overloading, the main loop in the code resembles the algorithm very closely. Now the conjugate gradient algorithm can be tested to solve a linear system with a 300 by 300 Hilbert coefficient matrix (which is symmetric and positive definite) and a known solution vector:

```

int main() {
    int n = 300;
    Mtx a(n, n);                      // n by n Hilbert matrix
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) a[i][j] = 1/(i + j + 1.0);

    Vcr t(n);                          // exact solution vector of size n
    Vcr x(n);                          // initial guess and solution vector
    for (int i = 0; i < n; i++)        // true solution
        t[i] = 1/(i + 3.14);

    int iter = 300;
    double eps = 1.0e-9;

    int ret = a.CG(x, a*t, eps, iter); // call CG algorithm
    if (ret == 0) cout << "CG returned successfully\n";
    cout << iter << " iterations are used in CG method.\n";
    cout << "Residual = " << eps << ".\n";
    cout << "Two-norm of exact error vector = "
        << (x - t).twonorm() << '\n' ;
}

```

The output of this program is:

```

CG returned successfully
11 iterations are used in CG method.
Residual = 1.31454e-09.
Two-norm of exact error vector = 4.44765e-06

```

The maximum number of iterations allowed can be taken to be the dimension of the matrix. In Chapter 11, this algorithm is used to solve matrix equations with real and complex matrices in different precisions and different matrix storage formats.

## 6.7 Exercises

- 6.7.1. Test and run the class for complex numbers defined in §6.1. Add more operations such as division, complex conjugate, complex equality operator `==` and inequality operator `!=`, prefix and postfix operators `--`, modulus, argument, and  $n$ th roots.
- 6.7.2. Modify the binary `+` and `-` operator functions for *Cmpx* as defined in §6.1 so that they have prototype:

```
Cmpx operator+(const Cmpx&, const Cmpx&);
Cmpx operator-(const Cmpx&, const Cmpx&);
```

The only difference is that pass by reference is now used in argument passing. Define these two functions and make sure your definitions do not cause unwanted side effects.

- 6.7.3. Test and run the class for vectors defined in §6.3. Then comment out the user-defined copy constructor and copy assignment and try to add two vectors  $v = v1 + v2$ . What would happen to vector  $v1$  in this vector addition? Could any other vector operations also be affected by depending on the compiler-generated copy constructor and copy assignment?
- 6.7.4. Redefine the class for vectors defined in §6.3 so that it provides range checking for vector indexes. That is, if  $v$  is a vector, then accessing  $v[i]$  with  $i < 0$  or  $i \geq v.size()$  will give a run-time error and exit the program.
- 6.7.5. Define a new class called *Cvec* for vectors of complex numbers using the simple definition in §6.1 and Exercise 6.7.1 for complex numbers. Provide basic operations (similar to those in §6.3 for real vectors) for complex vectors such as vector addition, vector multiplication, vector-scalar multiplication, and dot product. Note that the dot product of two complex vectors includes complex conjugation of the second vector.
- 6.7.6. Test and run the class for matrices defined in §6.3. Add more operations such as matrix-matrix multiplication, matrix-scalar multiplication, and one, maximum, and Frobenius matrix norms.

- 6.7.7. Implement the Gauss quadrature in Exercise 5.10.11 as the dot product of two vectors; one vector represents the weights and the other represents the function values at quadrature points. Evaluating definite integrals as dot products enables one to deal easily with integrals with the integrand being the product or sum of several functions (see §7.7).
- 6.7.8. Following the idea of Exercise 6.7.7, implement the Gauss–Lobatto quadrature problem in Exercise 5.10.12 as evaluating dot products on subintervals of a given interval  $[a, b]$ .
- 6.7.9. Mimic the class *tiny* defined in §6.4 to define a class for 6-bit non-negative integers that can mix freely with integers in arithmetic operations. Provide range checking when assigning an *int* to a variable of such a class and the conversion operator from it to *int*.
- 6.7.10. Test and run the deferred-evaluation operator overloading code for vector *saxpy* operations as defined in §6.5. Compare the run-time on your machine for the straightforward operator overloading as defined in §6.3, the deferred-evaluation operator overloading as defined in §6.5, and the traditional procedural-style function call *saxpy()* as defined in §6.5.
- 6.7.11. Extend the vector class defined in §6.5 using deferred-evaluation operator overloading, so that it will include efficient vector addition and scalar-vector multiplication (without any temporary vector objects and with a minimum number of loops).
- 6.7.12. Define a matrix class based on deferred-evaluation operator overloading as discussed in §6.5 to support efficient matrix-vector *gaxpy* (general  $A \times$  plus  $y$ ) operation  $z = A * x + y$ , where  $x, y, z$  are vectors and  $A$  a matrix. The implementation should not use any temporary vector objects and should apply only a minimum number of loops.
- 6.7.13. Apply the conjugate gradient algorithm to solve a system of linear equations with a symmetric and positive definite coefficient matrix, for example, with a Hilbert matrix of dimension 500 and a given right-hand side vector. Compute the two-norm of the residual of the numeric solution to check its accuracy.

# 7

## Templates

C++ provides a mechanism, called *template*, that allows a type to be a parameter in the definition of a class or function. For example, a class template allows a uniform and general definition for vectors whose elements can be *int*, *float*, *double*, *long double*, and so on. A function template permits its arguments and return value to have type parameters. This is different from function overloading that requires several similar definitions to be given.

A template is often called a parameterized type that directly supports generic programming. What a template provides is referred to as compile-time polymorphism, since the compiler will generate different forms of a template according to the types of the actual arguments used. As function overloading, class and function templates impose no run-time overhead.

Besides talking about class and function templates, this chapter also presents standard libraries on complex numbers, *valarrays*, and numeric algorithms. Surprisingly, templates can be used to improve run-time efficiency. Several efficient template techniques are given. The chapter ends with a section on polynomial interpolation.

### 7.1 Class Templates

The keyword *template* is used when defining a template class or function. For example, a simple template class for vectors with a type parameter *T* can be declared as

```

template<class T> class Vcr {
    int lenth;                // number of entries
    T* vr;                    // entries of vector
public:
    Vcr(int, const T* const); // constructor
    Vcr(int = 0, T = 0);      // constructor
    Vcr(const Vcr&);          // copy constructor
    ~Vcr(){ delete[] vr; }    // destructor

    int size() const { return lenth; } // number of entries
    Vcr& operator=(const Vcr&);        // assignment
    T& operator[](int i) { return vr[i]; } // subscripting
    Vcr& operator+=(const Vcr&);        // v += v2
    T maxnorm() const;                 // maximum norm

    template<class S>                // dot product
    friend S dot(const Vcr<S>&, const Vcr<S>&);
};

```

前缀

The `template<class T>` **prefix** specifies that a template is being declared with the template type parameter `T`. The scope of `T` extends to the end of the declaration. Here `T` means a type name that need not be a *class*. If one prefers, a template can be equivalently declared using the keyword **typename**:

```
template<typename T> class Vcr; // a template declaration
```

Notice the friend `dot()` is itself a template with template parameter `S`.

Once members and friend of this template class are defined, the compiler can automatically generate different classes. They can be used as

```

Vcr<double> dv(10);           // vector of 10 doubles,
Vcr<int> iv;                  // empty vector of int
Vcr<float*> fv(15);           // a vector of 15 (float*)s

```

```

double* b = new double [10];
for (int i = 0; i < 10; i++) b[i] = i;
Vcr<double> dv2(10, b);       // a vector of 10 doubles
dv2 += dv;                   // add and assign
double c = dot(dv,dv2);       // dot product of two vectors

```

Note that the type parameter is replaced by specific type names bracketed by `< >` and is put after the template class name. The compiler automatically generates definitions for classes `Vcr<double>`, `Vcr<int>`, and `Vcr<float*>` from the class template `Vcr<T>`. In particular, the compiler-generated class `Vcr<double>` is a class for vectors whose entries are doubles and works exactly like the class `Vtr` defined in §6.3. The use



of a template class does not impose any run-time overhead compared to an equivalent “hand-written” class such as *Vtr*, although certain compilation time is needed to generate specific classes from a class template. Such compilation time should be in general less than compiling different versions of a class for different element types when not using a template. Besides, templates can make user code shorter and have space efficiency.

It may sometimes be a good idea to debug a particular class such as *Vtr* before turning it into a class template such as *Vcr*<*T*>. In this way, many design problems and code errors can be handled in the context of a concrete example, which is generally easier than dealing with a general template class. However, for experienced programmers, defining a template directly may save a lot of code development time.

### 7.1.1 Member and Friend Definitions

Member functions of a template class are themselves templates. In the case of *Vcr*<*T*>, its member functions are templates parameterized by the parameter of the template class. (In general, a member function or friend can be parameterized by other parameters; see §7.2.5 and §7.2.6) They can be defined either inside the class declaration like the destructor and subscript operator above or outside as templates. For example,

```
template<class T> Vcr<T>::Vcr(int n, const T* const abd) {
    vr = new T [lenth = n];
    for (int i = 0; i < lenth; i++) vr[i] = *(abd + i);
}

template<class T> Vcr<T>& Vcr<T>::operator=(const Vcr& v) {
    if (this != &v) {
        if (lenth != v.lenth) cout << "bad assignment of vector";
        for (int i = 0; i < lenth; i++) vr[i] = v[i];
    }
    return *this;
}

template<class T> T Vcr<T>::maxnorm() const {
    T nm = abs(vr[0]);
    for (int i = 1; i < lenth; i++) nm = max(nm, abs(vr[i]));
    return nm;
}

template<class T> // dot product
T dot(const Vcr<T>& v1, const Vcr<T>& v2) {
    if (v1.lenth != v2.lenth) cout << "bad vector sizes\n";
    T tm = v1[0]*v2[0];
```

```

    for (int i = 1; i < v1.length; i++) tm += v1[i]*v2[i];
    return tm;
}

```

Other member functions can be defined similarly. Although `dot<S>` is declared using parameter `S`, it can be defined using any parameter such as `T`.

Within the scope of `Vcr<T>`, qualification with `<T>` to refer to `Vcr<T>` is redundant and can be omitted, although it is not an error to keep the redundancy. So the constructor and assignment operator can be alternatively defined as

```

template<class T> Vcr<T>::Vcr<T>(int n, const T* const abd) {
    vr = new T [length = n];
    for (int i = 0; i < length; i++) vr[i] = *(abd + i);
}

template<class T> Vcr<T>& Vcr<T>::operator=(const Vcr<T>& v){
    if (this != &vec) {
        if (length != v.length) cout << "bad assignment of vector";
        for (int i = 0; i < length; i++) vr[i] = v[i];
    }
    return *this;
}

```

However, for friends and ordinary function templates, the qualification with `<T>` is not redundant and must not be omitted.

### 7.1.2 Template Instantiation

The process of generating a class declaration from a template class and a specific template argument is often called template instantiation. A version of a template for a particular template argument is called a template specialization. For example, for the program,

```

int main() {
    Vcr<float> fv(20);
    Vcr<double> dv(500);
    Vcr<double> dv2(500, 3.14);
    dv += dv2;
}

```

the compiler will generate declarations for `Vcr<float>` and `Vcr<double>`, for their constructors and destructors, and for the operator `Vcr<double>::operator+=(const Vcr<double> &)`. Other member and nonmember functions are not used and should not be generated. Specializations are generated when they are used. The user can also force an *explicit instantiation*:

```
template class Vcr<long double>;           // a class
template short& Vcr<short>::operator[] (int); // a member fcn
```

When a class template is explicitly instantiated, every member and friend is also instantiated.

### 7.1.3 Template Parameters

A template can take several parameters and a template parameter can be used in the definition of subsequent template parameters. For example,

```
template<class C, class T, C val, int i> class Sometype {
    T v[i];
    int sz;
    C d = val;
public:
    Sometype(): sz(i) {cout << "member d is:" << d << '\n';}
};
```

```
Sometype<int, double, 3, 25> di32;
```

Here *val* of type *C* and *i* of type *int* are also parameters of the template class. Such parameters can be a constant expression, the address of an object or function with external linkage, or a nonoverloaded pointer to a member. It is an error if their values can not be determined at compile time.

### 7.1.4 Type Equivalence

Given a template class, different types (specializations) can be generated by supplying different template arguments. For example,

```
Vcr<long double> ldv;
Vcr<unsigned int> uiv(33);
Vcr<int> iv(33);
```

```
typedef unsigned int Uint;
Vcr<Uint> uintv(33);
```

The classes *Vcr<int>* and *Vcr<unsigned int>* are different types, while *Vcr<Uint>* and *Vcr<unsigned int>* are the same, since *typedef* simply creates a type synonym but does not introduce a new type.

### 7.1.5 User-Defined Specializations

A close look at the template class *Vcr<T>* and the friend *dot<T>()* reveals that it does not work correctly when the type parameter *T* is a complex

number, since the maximum-norm function `maxnorm()` should return a real number and the dot-product function `dot<T>()` should include complex conjugation. This could be fixed by including conjugation even for dot products of real numbers and letting `maxnorm()` always return a long double. But this would sacrifice efficiency. A user-defined specialization is a way to solve this problem. Now I define a version (a specialization) of the template class `Vcr<T>` when its elements are complex numbers; the C++ standard library `<complex>` (defined as a class template itself; see §7.4) is used in this definition.

```
// ***** a specialization of Vcr<T> for complex numbers
template<class T> class Vcr< complex<T> >{ // use <complex>
    int lenth;
    complex<T>* vr;
public:
    Vcr(int, const complex<T>* const);    // constructor
    Vcr(int = 0, complex<T> = 0);        // constructor
    Vcr(const Vcr&);                      // copy constructor
    ~Vcr() { delete[] vr; }              // destructor

    int size() const { return lenth; }
    complex<T>& operator[](int i) const { return vr[i]; }
    Vcr& operator+=(const Vcr&);          // add-assign
    Vcr& operator=(const Vcr&);           // assignment
    T maxnorm() const;                   // maximum norm

    template<class S> friend complex<S>    // dot product
    dot(const Vcr<complex<S> >&, const Vcr<complex<S> >&);
};
```

Note that this specialization is itself a template class; such a specialization is called a *partial specialization*. The `template<class T>` prefix specifies that this template class has a type parameter *T*. The `<complex<T>>` after the template class name means that this specialization is to be used when the template argument in `Vcr<T>` is a complex number. Its members and nonmembers must be redefined to accommodate special situations of vectors of complex numbers. They may be defined as

```
template<class T>
Vcr< complex<T> >::Vcr(int n, const complex<T>* const abd) {
    vr = new complex<T> [lenth = n];
    for (int i = 0; i < lenth; i++) vr[i] = *(abd + i);
}

template<class T>
Vcr< complex<T> >::Vcr(const Vcr & vec) { // copy constructor
```

```

    vr = new complex<T> [lenth = vec.lenth];
    for (int i = 0; i < lenth; i++) vr[i] = vec[i];
}

template<class T>
T Vcr< complex<T> >::maxnorm() const {    // maximum norm
    T nm = abs(vr[0]);
    for (int i = 1; i < lenth; i++)
        nm = max(nm, abs(vr[i]));        // use <algorithm>
    return nm;
}

template<class T>                                // dot product
complex<T> dot(const Vcr<complex<T> >& v1,
               const Vcr<complex<T> >& v2) {
    if (v1.lenth != v2.lenth) cout << "bad vector sizes\n";
    complex<T> tm = v1[0]*conj(v2[0]);      // conjugate of v2
    for (int i =1; i <v1.lenth; i++) tm += v1[i]*conj(v2[i]);
    return tm;
}

```

The prototypes and definitions for *maxnorm()* and *dot()* are very different now from the previous ones.

Now we can write the following function *main()* to test it.

```

int main(){
    int n = 300;
    complex<double>* aa = new complex<double> [n];
    for (int j = 0; j < n; j++) aa[j] = complex<double>(5, j);
    Vcr< complex<double> > v1(n, aa) ;           // vector v1

    Vcr< complex<double> > v2(n);                // vector v2
    for (int j = 0; j < n; j++) v2[j] =complex<double>(2, 3+j);

    cout << "norm = " << v1.maxnorm() << '\n'; // max norm
    cout << "dot = " << dot(v1,v2) << '\n'; // dot product
}

```

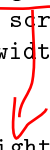
Here is another example of partial specialization:

```

template <int height, int width> class screen {
    // a class for screen with a height and width
    // height and width are two template parameters
};

template <int height> class screen<height, 80> {
    // a partialization for screens with 80-character width

```



```

    // which may be implemented more efficiently
    // Now only height is parameter, width = 80.
};

```

A complete specialization is one that does not take any type parameters. For example, a complete specialization for  $Vcr<T>$  is:

```

template<> class Vcr< complex<double> >{
    int lenth;                                // number of entries
    complex<double>* vr;                      // entries of vector
public:
    Vcr(int, complex<double>*);
    Vcr(int = 0, complex<double> = 0);
    Vcr(const Vcr&);                          // copy constructor
    ~Vcr() {delete[] vr; }                   // destructor

    Vcr& operator=(const Vcr&);
    int size() { return lenth; }
    complex<double>& operator[](int i) const { return vr[i]; }
    Vcr& operator+=(const Vcr&);              // add-assign
    double maxnorm() const;                  // maximum norm

    template<> friend                          // dot product
    complex<double> dot(const Vcr&, const Vcr&);
};

```

The `template<>` prefix specifies that this class is a specialization for  $Vcr<T>$  that does not take any type parameters. It is a complete specialization. The same explanation holds for template function `dot()`. All members and friends must be defined for a specialization. However, only two definitions are presented here (the rest are omitted to save space):

```

Vcr< complex<double> >&
Vcr< complex<double> >::operator+=(const Vcr& vec) {
    if (lenth != vec.lenth ) cout << "bad vector sizes\n";
    for (int i = 0; i < lenth; i++) vr[i] += vec[i];
    return *this;
}

template<>
complex<double> dot(const Vcr<complex<double> >& v1,
                   const Vcr<complex<double> >& v2) {
    if (v1.lenth != v2.lenth ) cout << "bad vector sizes\n";
    complex<double> tm = v1[0]*conj(v2[0]); // conjugation
    for (int i = 1; i < v1.lenth; i++) tm += v1[i]*conj(v2[i]);
    return tm;
}

```

All specializations of a template must be declared in the same namespace as the template itself. If a specialization is explicitly declared such as `Vcr<complex<T>>`, it (including all of its members) must be defined somewhere. In other words, the compiler will not generate definitions for it.

Similarly, a partial specialization for vectors of pointers can be defined as

```
// ***** a specialization of Vcr<T> for pointers
template<class T> class Vcr<T*>{
    int lenh;                // number of entries
    T** vr;                  // entries of vector
public:
    Vcr(int, T**);
    Vcr(const Vcr&);          // copy constructor
    T* &operator[](int i) const { return vr[i]; }
    Vcr& operator=(const Vcr&);
    // ... other members and definitions
};
```

Specialization is a way of specifying alternative and often efficient implementations for specific type parameters. However, the user interface is not changed; in other words, with or without the specializations, the user still uses a *Vcr* of complex numbers and a *Vcr* of pointers the same way. See Exercise 7.9.4 for an alternative approach to specializing vector *maxnorm()*.

### 7.1.6 Order of Specializations

One specialization is more specialized than another if one argument list that matches its specialization pattern also matches the other but not vice versa. For example,

```
template<class T> class Vcr;                // general
template<class T> class Vcr< complex<T> >;
    // specialized for complex numbers
template<> class Vcr< complex<float> >;
    // specialized for single precision complex numbers
```

In the above, the second class is more specialized than the first one, and the third is more specialized than the second. The most specialized version will be preferred over others in declarations of objects, pointers, and in overload resolution. For example, in the declaration,

```
Vcr< complex<double> > cdv(100);
```

the second class is preferred over the first and general class.

## 7.2 Function Templates

A function template defines a function that takes type parameters. In §7.1, the basic idea of a function template is illustrated through a dot product example `dot<T>()` (which is a function template that happens to be a friend of a template class) in the context of vectors. This section presents more details on function templates.

Assume we want to sort a general array of elements in increasing order; a sort function can be defined as a function template:

```
template<class T> void sort(T*, int); // function template

void f(int* vi, double* vd, string* vs, int n) {
    sort(vi, n);           // sort(int*, int), T = int
    sort(vd, n);           // sort(double*, int), T = double
    sort(vs, n);           // sort(string*, int), T = string
}
```

When a function template is called, the type of the function arguments determines which version (specialization) of the template is used. In other words, the template arguments are deduced from the function arguments. For example, in the call `sort(vi, n)`, the template argument is deduced as  $T = \text{int}$  from the type of `vi`.

Based on the shell sort algorithm [Knu98, Str97], the sort function template can be defined as

一般需要重载该运算符 `< / ==`

```
template<class T> void sort(T* v, int n) {
    for (int gap = n/2; gap > 0; gap /= 2)
        for (int i = gap; i < n; i++)
            for (int j = i - gap; j >= 0; j -= gap)
                if (v[j + gap] < v[j]) {
                    T temp = v[j];           // swap v[j] and v[j+gap]
                    v[j] = v[j + gap];
                    v[j + gap] = temp;
                }
}
```

Note that the operator `<` is used for comparison. However, not every type has such an operator. Improvements are given in §7.2.3 and §7.2.4. Another way is to overload the operator `<`. For example, an array of `point2d` can be sorted according to its  $x$ -coordinate as

```
struct point2d {
    double x, y;
};

inline bool operator<(const point2d& p, const point2d& q) {
```



```

    return p.x < q.x;        // compare x coordinates
}

int main() {
    const int n = 10000;
    point2d a[n];
    for (int i = 0; i < n; i++) {
        a[i].x = i*(5 - i); a[i].y = 5/(i+2.3) - 1;
    }
    sort(a, n);              // sort array a of n elements
}

```

Compare this definition with C-style polymorphic functions that rely on a pointer to function for comparison (see Exercise 3.14.24 or the book [Str97]). This template definition `sort<T>()` is shorter and most likely faster since function calls through pointers are hard to be inlined. Running times for this function template and the shell sort function `shsort()` defined in Exercise 3.14.24 for the array of 10000 *point2d* on a Pentium PC are 70 and 109 seconds, respectively. They are 27 and 86 seconds on a SUN Ultra2 workstation, 10 and 19 seconds on a Pentium II PC, and 18 and 30 seconds on an SGI workstation, respectively. On these four machines, this function template is much faster.

### 7.2.1 Function Template Parameters

推断

A compiler can **deduce** the types of the parameters of a function template, provided the function argument list identifies them uniquely. However, when a template parameter can not be deduced from the template function arguments, it must be specified explicitly within a pair of angle brackets:

```

template<class T> T* f(T);

int i = 5;
double d = 3.4;
f(i);                // template parameter T = int
f(5);                // template parameter T = int
f(d);                // template parameter T = double

template<class T, class S> T g(S);

g<int, double>(d);    // T = int, S = double
g<double>(i);         // T = double, S = int
g(i+d);              // error: can not deduce T and S
g<double, double>(i+d); // T = double, S = double

```

As with default function arguments, only trailing template parameters can be omitted. The call  $g<double>(i)$  is equivalent to  $g<double, int>(i)$  since the compiler can deduce  $S = int$  by the type of  $i$ . However, the call  $g(i + d)$  is ambiguous since there is no way to deduce template parameters  $T$  and  $S$  from it.

Note that class template parameters are not allowed to be deduced and must be specified explicitly since ambiguities arise for classes with multiple constructors.

### 7.2.2 Function Template Overloading

As functions can be overloaded, function templates can also be overloaded. For example,

```
template<class T> T abs(T);           // version 1, template
template<class T> T abs(complex<T>); // version 2, template
double abs(double);                 // version 3, ordinary fcn

abs(2.0);                           // abs(double), version 3
abs(2);                             // abs<int>(int), version 1
complex<double> z(2,3);
abs(z);                             // abs<double>(complex<double>)
```

In the case of more than one overloaded function template or specialization, the most specialized one will be chosen and used according to the arguments of the function call. The detailed overloading resolution rules are as follows.

1. Find the set of function template specializations that match the function call. For the function call  $abs(z)$  above, the specializations  $abs<double>(complex<double>)$  (version 2 with  $T = double$ ) and  $abs<complex<double>>(complex<double>)$  (version 1 with  $T = complex<double>$ ) are candidates.
2. If more than one template function can be called, consider the most specialized ones. For the call  $abs(z)$ , the specialization  $abs<double>(complex<double>)$  (version 2) is preferred over the specialization  $abs<complex<double>>(complex<double>)$  (version 1), because any call that matches  $abs<complex<T>>(complex<double>)$  (version 2) also matches  $abs<T>(complex<double>)$  (version 1).
3. Do overload resolution for this set of template function specializations and ordinary functions. If a template function argument has been determined by template argument deduction (§7.2.1), that argument can not have standard type conversions, promotions, or user-defined conversions. For the call  $abs(2)$ ,  $abs<int>(int)$  is an exact match (without type conversions, etc.) and preferred over  $abs(double)$ .

4. If an ordinary function and a template specialization are equally good matches, the function is preferred. For example, the function call `abs(double)` is preferred over `abs<double>(double)` (version 1 with  $T = \text{double}$ ) for the call `abs(2.0)`.
5. If more than one equally good match is found, the call is ambiguous and an error. If no match is found, the call is also an error.

The following is another example.

```
template<class C> C absmax(C a, C b) {
    return (abs(a) > abs(b)) ? a: b;
}

absmax(-1, 2);           // absmax<int>(1, 2)
absmax(2.7, 5.6);        // absmax<double>(2.7,5.6)
absmax(5, 6.9);          // ambiguous, absmax<double>(5,6.9)
                        // or absmax<int>(5,6.9)?
```

The call `absmax(5, 6.9)` is ambiguous and illegal, since no standard conversion is made for its arguments. This ambiguity can be resolved either by explicit qualification:

```
absmax<double>(5, 6.9);
```

or by defining an ordinary function, to which standard type conversion applies:

```
inline double absmax(double a, double b) {
    return absmax<double>(a,b);
}
```

If a function template also takes basic types as its arguments, for example,

```
template<class T, class S, class R> R f(T& t, S s, double d);
```

then standard conversions apply to the basic type argument  $d$ , although no standard conversion can be implied for the arguments  $t$  and  $s$ .

### 7.2.3 Specializations

The sort program given early in this section does not sort arrays of `char*` correctly since it will compare the addresses of the first `char` for each element of the array. **Specialization can be used to define a comparison function that compares different types of elements correctly.** We now use the standard library `<vector>`, defined as a class template (see §10.1.1), for a dynamic array of elements. For example,

```

template<class T> bool less(T a, T b) {
    return a < b;           // a general template
}

template<> bool less<const char*>(const char* a,
                                const char* b) {
    return strcmp(a,b);    // specialization
}                               // use string-compare in <cstring>

template<class T> void sort(vector<T>& v) {
    const unsigned long n = v.size();
    for (int gap = n/2; gap > 0; gap /= 2)
        for (int i = gap; i < n; i++)
            for (int j = i - gap; j >= 0; j -= gap)
                if (less(v[j+gap], v[j])) swap(v[j], v[j+gap]);
}

```

Here the function template `swap()`, defined in the library `<algorithm>`, is used to swap the values of two variables. As for class templates, the `template<>` prefix implies a specialization that does not have a type parameter. The `<const char*>` after the template function name means that this specialization is to be used when the template argument  $T = \text{const char*}$ . Now the function template `sort<T>()` should sort numbers and strings correctly. Since the template argument can be deduced from the function argument list, this specialization can also be written as

```

template<> bool less(const char* a, const char* b) {
    return strcmp(a,b);    // specialization, using <cstring>
}

```

If complex numbers are going to be compared according to their absolute values, another specialization can be defined as

```

template<class T> bool less(complex<T> a, complex<T> b) {
    return abs(a) < abs(b); // specialization, use <complex>
}

```

With different specializations of template `less<T>()`, a vector of strings, a vector of complex numbers, or a vector of `point2d` can be sorted in increasing order or any other order defined by the specializations.

Another example of function template specialization is the useful vector dot product, applied to real and complex arrays:

```

template<class T> T dot(T* a, T* b, int n) {
    T init = 0;
    for (int i = 0; i < n; i++) init += (*a++)*(*b++);
    return init;
}

```

```
template<class T> // specialization for complex vectors
complex<T> dot(complex<T>* a, complex<T>* b, int n) {
    complex<T> init = 0;
    for (int i = 0; i < n; i++) init += (*a++)*conj(*b++);
    return init;
}
```

#### 7.2.4 *Class Templates as Function Template Parameters*

When comparing vectors of *char* or other types according to different criteria such as case-sensitive and case-insensitive comparisons, **it is convenient to pass the criterion as a class template to the compare-function template.**

For example, define a function template *compare*<*T*>():

```
template<class T, class S>
int compare(const vector<T>& v1, const vector<T>& v2) {
    for (int i = 0; i < v1.size() && i < v2.size(); i++)
        if (!S::eq(v1[i], v2[i])) return S::st(v1[i], v2[i]);
    return v2.size() - v1.size();
}
```

Here the template parameter *S* can be a class whose member functions *S* :: *eq*() and *S* :: *st*() are defined to represent the comparison or sorting criteria. For example,

```
template<class T> class Ncomp {           // normal compare
public:
    static int eq(T a, T b) { return a == b; } // equal
    static int st(T a, T b) { return a < b; }  // smaller than
};

class Nocase {                           // compare by ignoring case
public:
    static int eq(char a, char b) {         // equal
        return toupper(a) == toupper(b);   // #include <ctype.h>
    }
    static int st(char a, char b) {         // smaller than
        return toupper(a) < toupper(b);     // case-insensitive
    }
};
```

The class *Nocase* changes characters into their corresponding uppercase equivalents (using *std* :: *toupper*()) before comparing. Each class generated from a class template gets a copy of each static member of the class template. Now vectors can be compared according to rules specified by the template argument:

```
void f(vector<char> v1, vector<char> v2) {
    compare<char, Ncomp<char> >>(v1, v2); // case-sensitive
    compare<char, Nocase>(v1, v2);        // case-insensitive
}
```

Passing comparison criteria as template parameters has the advantage that it allows several operations to be passed as a single argument and the comparison operators *eq()* and *st()* are trivial to be inlined. This should increase run-time efficiency compared to C- or FORTRAN-style passing function pointers; see §7.2 and §7.6 for run-time comparisons.

Default template arguments can be used as well:

```
template<class T, class S = Ncomp<T> > // default parameter
int compare(const vector<T>& v1, const vector<T>& v2) {
    // define the function template as before
}
```

Then the normal comparison criterion need not be specified each time it is used:

```
void g(vector<char> v1, vector<char> v2) {
    compare(v1, v2); // use Ncomp<char>
    compare<char, Nocase>(v1, v2); // use Nocase
}
```

Classes used as template parameters for expressing criteria or policies are often called “*traits*”. See Exercise 7.9.4 for another example of traits.

### 7.2.5 Member Function Templates

A class or class template can have function members that are parameterized by general parameters. For example,

```
template<class S> class Cmpx {
    S re, im;
public:
    template<class T> Cmpx(const Cmpx<T>& c)
        : re(c.re), im(c.im) { }
};
```

The copy constructor of this class template is a template function that defines a type conversion from *Cmpx<T>* to *Cmpx<S>*. It can be used as

```
void f(Cmpx<float> cf) {
    Cmpx<double> cd = cf; // float to double conversion
}
```

### 7.2.6 *Friend Function Templates*

A friend function of a class or a class template may be an ordinary function, a function template, or an instantiation of a function template. In §7.1, the function template *dot*<*S*> () is declared to be a friend of class template *Vcr*<*T*>:

```
template<class T> class Vcr {
    // ... in addition to other members or friends

    template<class S>                // dot product
    friend S dot(const Vcr<S>&, const Vcr<S>&);
};
```

This means that all instantiations of the function template *dot*<*S*> () are friends to any instantiation of class template *Vcr*<*T*>. In particular, *dot*<*double*> () is a friend of class instantiations *Vcr*<*double*> and *Vcr*<*float*>.

It is also possible to first define or declare a function template such as *dot*<*T*> () and then declare an instantiation of it to be a friend of a class or class template such as class *Vcr*<*T*>. For example,

```
template<class T> class Vcr; // template forward declaration

template<class T>          // template forward declaration
friend T dot(const Vcr<T>&, const Vcr<T>&);

template<class T> class Vcr { // class template definition
    // ... in addition to other members or friends

    friend T dot<T>(const Vcr&, const Vcr&);
};
```

This means that only one corresponding instantiation of *dot*<*T*>() is a friend to each instantiation of *Vcr*<*T*>. In particular, *dot*<*double*>() is a friend of class *Vcr*<*double*>, but not *Vcr*<*float*>. Notice the instantiation syntax in the friend declaration *dot*<*T*>(). When it is omitted:

```
template<class T> class Vcr { // class template definition
    // ... in addition to other members or friends

    friend T dot(const Vcr&, const Vcr&);
};
```

this would be interpreted as declaring an ordinary function *dot*() to be a friend to all instantiations of class template *Vcr*<*T*>. It could be defined as

```
float dot(const Vcr<float> &v1, const Vcr<float> &v2) {
```

```

    if (v1.length != v2.length ) error("bad vector sizes");
    float tm = v1[0]*v2[0];
    for (int i = 1; i < v1.length; i++) tm += v1[i]*v2[i];
    return tm;
}

double dot(const Vcr<double>& v1, const Vcr<double>& v2) {
    // ...
}

```

Similarly, a class, a class template, or an instantiation of a class template can be a friend of another class or class template.

### 7.3 Template Source Code Organization

Specializations of class and function templates are generated only when needed (§7.1.2). **Thus the definition of templates must be known to the source code that uses the templates.** A trivial strategy is to put everything in one file so that the compiler can easily generate code as needed. But this is applicable only to small programs. There are two other ways of organizing the source code.

Consider the simple example for a function template that prints some user-specified (error) messages and terminates the program. Assume the definition of the template is put in a file called *error.cc*.

```

// file error.cc
#include <cstdlib>
#include <iostream>
template<class T> void error(const T & t) {
    std::cout << t << "\nProgram exited.\n";
    std::exit(1);
}

```

The first way is to include this template definition file in all files that use the template. For example,

```

// file f1.cc
#include "error.cc"
int* a = new (nothrow) int [1000000];
// return 0 when out of memory
if (!a) error("no more space.");

// file f2.cc
#include "error.cc"
int* b = new (nothrow) int [1000000];

```



```
if (!b) error(*b);
```

That is, the definition of the template and all declarations that it depends on are included in different compilation units. The operator *new* throws an exception *bad\_alloc* when there is no more memory available. However, using *(nothrow)* causes it to return 0 instead; see §9.5. The compiler will generate appropriate specializations according to how the template is used. This strategy treats templates the same way as inline functions.

The second way is to include only declarations of templates and explicitly *export* its definitions to different compilation units. Split the original *error.cc* file into two:

```
// file error2.h: only declarations of the template
template<class T> void error(const T & );

// file error2.cc: definition of template and export it
#include <cstdlib>
#include <iostream>
#include "error2.h"
export template<class T> void error(const T & t) {
    std::cout << t << "\nProgram exited.\n";
    std::exit(1);
}
```

Now the declaration of the template can be included in different files:

```
// file f3.cc
#include "error2.h"
int* a = new (nothrow) int [1000000];
if (!a) error("no more space.");

// file f4.cc
#include "error2.h"
int* b = new (nothrow) int [1000000];
if (!b) error(*b);
```

This strategy treats templates the same way as noninline (ordinary) functions. The template definition in file *error2.cc* is compiled separately, and it is up to the compiler to find the definition of the template and generate its specializations as needed. Note that the keyword *export* means accessible from another compilation unit. This can be done by adding *export* to the definition or declaration of the template. Otherwise the definition must be in scope wherever the template is used.

## 7.4 Standard Library on Complex Numbers

Complex numbers are provided as a template class in the C++ standard library `<complex>`. It is defined as

```
template<class T> class std::complex {
    T re, im;
public:
    complex(const T& r = T(), const T& i = T()); // constructor
    template<class S> complex(const complex<S>&); // constructor

    complex<T>& operator=(const T&);
    template<class S> complex<T>& operator=(const complex<S>&);
    // also operators +=, -=, *=, /=

    T real() const { return re; }
    T imag() const { return im; }
};
```

Notice the two template member functions above, which enable construction and assignment from other types of complex numbers, for example, from `complex<float>` to `complex<double>`. However, specializations will be defined to restrict implicit conversions, for example, from `complex<double>` to `complex<float>`, in order to avoid truncation.

Other functions acting on complex numbers can be defined as

```
template<T> complex<T> operator+(const complex<T>&,
                                const complex<T>&);
template<T> complex<T> operator+(const T&, const complex<T>&);
template<T> complex<T> operator+(const complex<T>&, const T&);
// also binary operators: -, *, /, ==, !=

template<T> complex<T> operator-(const complex<T>&);
// also unary operator: -

template<T> T real(const complex<T>&); // real part
template<T> T imag(const complex<T>&); // imaginary part
template<T> complex<T> conj(const complex<T>&); // conjugate

template<T> T abs(const complex<T>&); // absolute value
template<T> T arg(const complex<T>&); // argument
template<T> complex<T> polar(const T& abs, const T& arg);
// polar form (abs, arg)
template<T> T norm(const complex<T>&); // square of abs()

template<T> complex<T> sin(const complex<T>&); // sine
```

```
// also: cos, tan, sqrt, sinh, cosh, tanh, exp,
//      log, log10, asin, acos, atan

template<T> complex<T> pow(const complex<T>&, int);
template<T> complex<T> pow(const complex<T>&, const T&);
template<T> complex<T> pow(const T&, const complex<T>&);
template<T> complex<T> pow(const complex<T>&,
                           const complex<T>&);
```

The function *polar()* constructs a complex number given its absolute value and argument. Note the misleading function name of *norm()*, which gives the square of the absolute value of a complex number. The input and output operators `<<` and `>>` are also provided. Specializations for *complex<float>*, *complex<double>*, and *complex<long double>* are defined to prevent certain conversions and to possibly optimize the implementations. For example,

```
class complex<float> {
    float re, im;
public:
    complex(float r = 0.0, float i = 0.0);
    complex(const complex<float>&);
    explicit complex(const complex<double>&);
    explicit complex(const complex<long double>&);
    // ... other functions
};
```

The explicit constructors declared above prevent implicit conversions from *complex<double>* and *complex<long double>* to *complex<float>*. For example,

```
complex<float> cf(3,4);
complex<double> cd(5,6);
cd = cf;           // OK, implicit conversion
cf = cd;           // illegal, no implicit conversion
cf = complex<float>(cd); // OK, explicit conversion
```

## 7.5 Standard Library on *valarrays*

C++ provides a standard library called *<valarray>* for optimized vector arithmetics in high performance computing. Since the design priority of this library is speed of computing, other features such as flexibility, generality, and type checking are sacrificed. For example, it does not check to see if the sizes of two vectors match before adding or multiplying them, and does not provide operations for various norms of a vector and dot products. Matrices are not directly supported, but a user can manipulate matrices

and submatrices through the (one-dimensional) *valarray* mechanism. If your main concern is easy-to-use and easy-to-modify in a general setting, you may want to write your own library (see Chapter 11) to fit your need.

### 7.5.1 The Type *valarray*

Some members of the type *valarray* are defined as

```
template<class T> class std::valarray {
public:
    valarray();                // valarray with size = 0
    explicit valarray(size_t n); // valarray with size = n
    valarray(const T* p, size_t n); // size =n, elements *p
    valarray(const T& v, size_t n); // size =n, all entries =v
    valarray(const valarray&);    // copy constructor
    ~valarray();                 // destructor

    valarray& operator=(const valarray&); // copy assignment
    valarray& operator=(const T& v);      // all elements = v
    T operator[](size_t) const;          // subscript, rvalue
    T& operator[](size_t);               // subscripting

    // scalar-vector operations: +=, -=, *=, /=, ^=, &=,
    //                               |=, <=<=, >=>=
    valarray& operator+=(const T& v); // add v to every element

    // unary operator: -, +, ~, !
    valarray& operator-();             // unary minus

    size_t size() const;               // number of elements
    void resize(size_t n, const T& v= T()); // all entries = v
    T sum() const;                     // sum of all elements

    valarray cshift(int i) const;      // cyclic shift of entries
    valarray shift(int i) const;       // logical shift
    valarray apply(T f(T)) const;      // apply f to each entry
    valarray apply(T f(const T&)) const; // apply f()
};
```

Here *size\_t* is an unsigned integral type (possibly *unsigned long*) defined in `<stddef>` and used in the standard libraries. In functions *shift(i)* and *cshift(i)*, left shift of elements by *i* positions is performed when *i* > 0, and right shift when *i* < 0.

Here is an example for illustrating some usages of *valarray*.

```
double* pd = new double [50000];
```

```

for (int i = 0; i < 50000; i++) pd[i] = i*i;
valarray<double> v0(pd,50000);    // vector of 50000 doubles

int permu[] = {1,2,3,4,5};        // permutation
valarray<int> v(permu,5);          // vector of 5 int
valarray<int> v2 = v.shift(2);     // v2 = {3,4,5,0,0}
valarray<int> v3 = v.cshift(2);   // v2 = {3,4,5,1,2}, cyclic
valarray<int> v4 = v.shift(-2);    // right shift, v2={0,0,1,2,3}
valarray<int> v4 = v.cshift(-2);  // right shift, v2={4,5,1,2,3}

```

Other operators and functions are provided as nonmember functions:

```

// +, -, *, /, ^, &, |, <<, >>, &&, ||, ==, !=, <, >, <=, >=,
// sin, cos, tan, asin, acos, atan, atan2, sinh, cosh, tanh,
// pow, exp, log, log10, sqrt, abs
valarray<T> operator+(const valarray<T>&, const valarray<T>&);
valarray<T> operator+(const T&, const valarray<T>&);
valarray<T> operator+(const valarray<T>&, const T&);
valarray<T> abs(const valarray<T>&);

T min(const valarray<T>&);         // smallest value
T max(const valarray<T>&);         // largest value

```

Note the function *abs*() needs a specialization in order to apply to a *valarray* of complex numbers.

The output operator is not provided for *valarray* since different users do it differently, especially in different applications. Neither is the input operator provided.

### 7.5.2 Slice Arrays

Slices are introduced in *<valarray>* for manipulating matrices through one-dimensional arrays. A slice describes every *n*th element of some part of a *valarray*, for a positive integer *n*. It is defined as

```

class std::slice {
    // ... representation, starting index, size, stride
public:
    slice();
    slice(size_t start, size_t size, size_t stride);
    size_t start() const;    // index of first element
    size_t size() const;     // number of elements in slice
    size_t stride() const;   // entry i is at start()+i*stride()
};

```

A stride is the distance (in number of elements) between two adjacent elements in the slice. For example, the 4 by 3 matrix  $A$  can be laid out as a vector  $B$  of size 12 :

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \\ a_{30} & a_{31} & a_{32} \end{bmatrix},$$

$$B = \begin{matrix} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ [a_{00} & a_{01} & a_{02} & a_{10} & a_{11} & a_{12} & a_{20} & a_{21} & a_{22} & a_{30} & a_{31} & a_{32}]. \end{matrix}$$

Then  $slice(0, 3, 1)$  of vector  $B$  describes row 0 of matrix  $A$  (consisting of 3 elements starting from 0 with stride 1; that is,  $B[0] = a_{00}$ ,  $B[1] = a_{01}$ , and  $B[2] = a_{02}$ ) and  $slice(3, 3, 1)$  describes row 1. Similarly, the  $slice(0, 4, 3)$  of vector  $B$  describes column 0 of matrix  $A$  and  $slice(2, 4, 3)$  column 2. That is, a  $slice(s, z, d)$  defines a mapping from three nonnegative integers  $\{s, z, d\}$  into an index set  $I = \{i : i = s + j * d, j = 0, 1, \dots, z - 1\}$  that describes some elements  $v[i], i \in I$ , of a *valarray*  $v$ . In particular, a matrix can be represented through a *valarray* and its rows and columns can be described by slices. The idea is that a row or column of a matrix can be accessed by using one index  $j$  for  $j = 0, 1, \dots, z - 1$ , where  $z$  is the number of elements in the row or column. Note that the row by row layout has been used for representing a matrix through a *valarray*. FORTRAN-style column by column layout can be used similarly. However, it may be better to get used to the C++ style (row by row layout, array index starting from 0 instead of 1, and the symmetric notation  $m[i][j]$  instead of  $m(i, j)$ ) if one wants to program in C++ extensively.

A type called *slice\_array* is defined in `<valarray>` to refer to elements of a *valarray* described by slices:

```
template <class T> class std::slice_array {
private:
    slice s;
    valarray<T>* p;                                // implementation-dependent

    slice_array();                                // prevent construction
    slice_array(const slice_array&);              // prevent construction
    slice_array& operator=(const slice_array&);
public:
    void operator=(const valarray<T>&);
    void operator=(const T& t);                    // assign t to each entry
    void operator*=(const T&);                      // scalar-vector multiply
```

```
// also: +=, -=, /=, %=, ^=, &=, |=, <<=, >>=

~slice_array();
};
```

Since construction and copy operations are defined as private, a user can not directly create a *slice\_array*. Instead, subscripting a *valarray* is defined to be the way to create a *slice\_array* corresponding to a given slice:

```
template <class T> class std::valarray {
public:
    // in addition to other members
    valarray(const slice_array<T>&);
    valarray operator[](slice) const;
    slice_array<T> operator[](slice);

    valarray& operator=(const slice_array<T>&);
};
```

Once a *slice\_array* is defined, all operations on it go to the *valarray*:

```
void create(valarray<double>& vd) {
    slice s(1, vd.size()/2, 2);
        // slice for elements vd[1], vd[3]...
    slice_array<double>& sad = vd[s];
        // subscripting a valarray by a slice
    sad[0] = 3.1415926; // vd[1] = 3.1415926
    sad = 2.2;        // assign 2.2 to vd[1], vd[3], ...
    float a = sad[1]; // a = vd[3]
    float b = sad[2]; // b = vd[5]

    valarray<double> ods = vd[s];
        // ods has only odd-index entries of vd
}
```

Note that it is illegal to copy *slice\_arrays*. The reason for this is to allow alias-free optimizations and prevent side effects. For example,

```
slice_array<double> create2(valarray<double>& vd) {
    slice_array<double> sad0; // error, no default constructor

    slice s(1, vd.size()/2, 2);
        // slice for odd index entries of vd
    slice_array<double> sad1 = vd[s];
        // illegal, no copy construction

    sad1 = sad0; // illegal, no copy assignment
    return sad0; // illegal, can not make a copy
}
```

### 7.5.3 Generalized Slice Arrays

A slice can be used to describe rows and columns of a matrix and multi-dimensional arrays represented through a *valarray*. In some applications, submatrices need be manipulated. Generalized slices *gslice* and arrays *gslice\_array* are introduced for this purpose.

An  $n$ -slice or generalized slice *gslice* contains  $n$  different dimensions ( $n$  slices) and is defined as

```
class std::gslice {
    // ... representation, starting index, n sizes, n strides
public:
    gslice();
    gslice(size_t start, const valarray<size_t> nsize,
           const valarray<size_t> nstride);
    size_t start() const;           // index of first entry
    valarray<size_t> size() const;  // number of elements in all dimensions
    valarray<size_t> stride() const; // strides for all dimensions
};
```

That is, an  $n$ -slice  $gslice(s, z, d)$ , where  $z = \{z_k\}$ ,  $d = \{d_k\}$ ,  $0 \leq k < n$ , defines a mapping from  $(s, z, d)$  into an index set  $I = \{i : i = s + \sum_{k=0}^{n-1} j_k * d_k, \text{ where } j_k = 0, 1, \dots, z_k - 1\}$  that describes some elements  $v[i], i \in I$ , of a *valarray*  $v$ . For example, consider a 3 by 2 submatrix  $A_{32}$  (inside the box in the picture below) in the lower-right corner of a 4 by 3 matrix  $A$  represented row by row through a *valarray*  $B$ :

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \\ a_{30} & a_{31} & a_{32} \end{bmatrix},$$

$$B = \begin{matrix} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ a_{00} & a_{01} & a_{02} & a_{10} & a_{11} & a_{12} & a_{20} & a_{21} & a_{22} & a_{30} & a_{31} & a_{32} \end{matrix}.$$

Since every row of the submatrix  $A_{32}$  can be represented by a slice of size 2 and stride 1 and every column by a slice of size 3 and stride 3, a 2-slice can be defined to describe the submatrix:

```
size_t z[] = {2,3};           // slice(s,z[0],d[0]) for a row
size_t d[] = {1,3};           // slice(s,z[1],d[1]) for a column
{it valarray}<size_t> vz(z,2); // size vector
```



```
{\it valarray}<size_t> vd(d,2); // stride vector
gslice gs2(4,vz,vd);      // starting index is 4
```

Thus *gs2* defines a mapping from  $\{(4, vz, vd)\}$  into the index set  $I = \{i : i = 4 + j_0 + 3j_1, j_0 = 0, 1, j_1 = 0, 1, 2\}$  for the submatrix  $A_{32}$ . In particular, the subset  $\{i : i = 4 + j_0 + 3j_1, j_0 = 0, 1, j_1 = 0\} = \{4, 5\}$  describes row 0 of  $A_{32}$  (i.e.,  $B[4] = a_{11}, B[5] = a_{12}$ ), and  $\{i : i = 4 + j_0 + 3j_1, j_0 = 0, j_1 = 0, 1, 2\} = \{4, 7, 10\}$  describes column 0 of  $A_{32}$  (i.e.,  $B[4] = a_{11}, B[7] = a_{21}, B[10] = a_{31}$ ). The idea is that the submatrix  $A_{32}$  can now be conveniently accessed using two indexes  $j_0$  and  $j_1$  for  $j_0 = 0, 1, \dots, z_0 - 1$  and  $j_1 = 0, 1, \dots, z_1 - 1$ , where  $z_0$  is the number of elements in each row and  $z_1$  the number of elements in each column of the submatrix.

The template class *gslice\_array* is defined in the same way and offers the same set of members. This implies that a *gslice\_array* can not be constructed directly and can not be copied by the user. Instead, subscripting a *valarray* is defined to be the way to create a *gslice\_array* corresponding to a given *n*-slice:

```
template <class T> class std::valarray {
public:
    // in addition to other members
    valarray(const gslice_array<T>&);
    valarray operator[] (const gslice&) const;
    gslice_array<T> operator[] (const gslice&);

    valarray& operator=(const gslice_array<T>&);
};
```

#### 7.5.4 Mask Arrays and Indirect Arrays

A *mask\_array* is defined from a *valarray* *v* through a mask *valarray*<*bool*>, with *true* indicating an element of the *valarray* *v* is selected. Subscripting a *valarray* through a mask creates a *mask\_array*:

```
template <class T> class std::valarray {
public:
    // in addition to other members
    valarray(const mask_array<T>&);
    valarray operator[] (const valarray<bool>&) const;
    mask_array<T> operator[] (const valarray<bool>&);

    valarray& operator=(const mask_array<T>&);
};
```

Similarly, an *indirect\_array* is defined from a *valarray* *v* through an index array (a permutation) *valarray*<*size\_t*>. Subscripting a *valarray* through an index array *valarray*<*size\_t*> creates an *indirect\_array*:

```
template <class T> class std::valarray {
```

```

public:                // in addition to other members
    valarray(const indirect_array<T>&);
    valarray operator[] (const valarray<size_t>&) const;
    indirect_array<T> operator[] (const valarray<size_t>&);

    valarray& operator=(const indirect_array<T>&);
};

```

Here is an example on how to use a *mask\_array* and *indirect\_array*:

```

void g(valarray<double>& vd) {
    bool ba[4] = {false, true, false, true};
    valarray<bool> mask(ba,4);           // a mask
    valarray<double> vd2 = sin(vd[mask]); // vd2[0]=sin(vd[1])
                                           // vd2[1]=sin(vd[3])

    size_t ia[4] = {3, 1, 0, 2};
    valarray<size_t> index(ia,4);        // an index array
    valarray<double> vd3 = sin(vd[index]); //vd3[0]=sin(vd[3])
                                           // vd3[1] =sin(vd[1]), vd3[2]=sin(vd[0]), vd3[3]=sin(vd[2])
}

```

An irregular subset of an array can be selected by a *mask\_array* and a reordering of an array can be achieved by an *indirect\_array*. An index array must be a permutation; that is, it can not contain the same index more than once. For example, the behavior of the *indirect\_array* is undefined if the above index array is defined as

```
size_t ia[4] = {3, 3, 0, 2};
```

The same set of members for *mask\_array* and *indirect\_array* are defined as for *slice\_array*. In particular, *mask\_array* and *indirect\_array* can not be directly constructed or copied by a user.

## 7.6 Standard Library on Numeric Algorithms

The standard library *<numeric>* contains a few numeric algorithms for performing operations on sequences such as the built-in array (§3.3), *valarray* (§7.5), and *vector* (§10.1.1). Other standard algorithms such as sorting a sequence of elements and finding the maximum value of elements in a sequence are discussed in §10.2.

The numeric algorithms are function templates and can be summarized as

<i>accumulate()</i>	accumulate results on a sequence
<i>inner_product()</i>	accumulate results on two sequences
<i>partial_sum()</i>	generate a sequence of “partial sums”
<i>adjacent_difference()</i>	generate sequence of “adjacent differences”

### 7.6.1 *Accumulate*

The function template *accumulate*() calculates the sum of the elements in a sequence given a starting and an ending pointer (or iterator; see §10.1.2), and an initial sum. A general binary operation, instead of summation, can also be specified as the fourth argument. It is defined as

```
template<class In, class T>
T accumulate(In first, In last, T init) {
    while (first != last) init = init + *first++;
    return init;                // summation
}

template<class In, class T, class BinOp>
T accumulate(In first, In last, T init, BinOp op) {
    while (first != last) init = op(init, *first++);
    return init;                // general operation
}
```

An explanation of the meaning of *(\*p++)* is given in §3.10. We can use *accumulate*() to, for example, calculate the sum or product of elements or their absolute values of an array. It can also be used to calculate the one, two, and maximum norms (§4.1) of a vector. A general binary operation can be passed as a pointer to a function. It can be used as

```
double SumAbs(double a, double b) { return a + fabs(b); }

int main() {
    double ai[5];
    for (int i = 0; i < 5; i++) ai[i] = -1;

    double d = 0;
    d = accumulate(ai, ai + 5, d);
                        // sum by default, d = -5
    d = accumulate(ai, ai + 5, 0.0, SumAbs);
                        // sum of absolute values, d = 5
}
```

The first and second arguments of *accumulate*() must point to the first and the last-past-one elements of a sequence, respectively. With the operation *SumAbs*(), it actually computes the one-norm of a sequence. It can also be used to compute, for example, the sum of *x*-coordinates of an array of *point2d*:

```
struct point2d { double x, y; };

double Sumx(double a, point2d p) { return a + p.x; }
```

```
double f(point2d* p, int n) { // sum of x-coordinates
    return accumulate(p, p + n, 0.0, Sumx);
}
```

Note that the second version of the function template *accumulate()* takes a binary operation as its fourth argument. The fourth argument can be either a binary function such as *SumAbs()* above, or a class object with the function call operator *()* overloaded. For example,

```
template<class T> struct Multiply { // overload operator()
    T operator()(const T& a, const T& b) const { return a*b; }
};

int main() {
    double ai[5];
    for (int i = 0; i < 5; i++) ai[i] = -1;
    double d =
        accumulate(ai, ai + 5, 1.0, Multiply<double>()); //d=-1
}
```

Notice an object rather than a type is needed for the fourth argument of *accumulate()*. Thus *Multiply<double>()* is used instead of *Multiply<double>*. An object of a class with the function call operator *()* overloaded is called a *function-like object*, or a *function object*. Note that from the definition of *accumulate()*, the operator *()* is passed to its fourth argument *op* in such a way that *op* can be either a function or a function object. More examples on function objects are presented in §7.7 and §10.2.

Passing a function object can be more easily inlined than a call through a pointer to a function. In the following example, we measure the times they take in performing the same arithmetic operations.

```
template<class T> T SumFcn(T a, T b) { return a + b; }

template<class T> struct SumFO {
    T operator()(const T& a, const T& b) const {return a + b;}
};

int main() {
    int n = 10000;
    double* ai = new double [n];
    for (int i = 0; i < n; i++) ai[i] = i - 500;
    double d = 0;

    time_t tm1 = time(0); // #include <time.h>

    for (int i = 0; i < 10000; i++) // repeat 10000 times
        d = accumulate(ai, ai + n, 0.0); // use default sum
}
```

```

time_t tm2 = time(0);           // current time
cout << "time in default summation = "
    << tm2 -tm1 << " seconds" << '\n';

for (int i = 0; i < 10000; i++) // pass pointer to fcn
    d = accumulate(ai, ai + n, 0.0, SumFcn);
time_t tm3 = time(0);           // measure time
cout << "time in passing function = "
    << tm3 -tm2 << " seconds" << '\n';

for (int i = 0; i < 10000; i++) // pass fcn object
    d = accumulate(ai, ai + n, 0.0, SumF0<double>());
time_t tm4 = time(0);           // measure time
cout << "time in passing fcn object = "
    << tm4 -tm3 << " seconds" << '\n';
}

```

On a Pentium PC running Linux, the running times are 24, 47, and 33 seconds, respectively, while on an SGI PowerChallenge they are 8, 16, and 14 seconds, and on a SUN Ultra2 workstation 10, 32, and 16 seconds. Thus it can be more efficient to use a function object or a template argument than passing a function pointer to another function. This implies that the FORTRAN- and C-style passing functions as arguments to other functions should be avoided if possible when a large number of such function calls are needed. However, both of them can be much less efficient than plain function calls. This suggests that a user may wish to define his or her own version of *accumulate()* and other functions when portability and readability are not as important as efficiency. For example,

```

template<class In, class T>      // user-defined version
T accumulate_multiply(In first, In last, T init) {
    while (first != last) init *= *first++; // multiply
    return init;
}

template<class In, class T>      // user-defined version
T accumulate_abs(In first, In last, T init) {
    while (first != last) init += abs(*first++);
    return init;                // sum of absolute values
}

```

These user-defined function templates do not require a function object or pointer to a function passed as an argument and should be more efficient than the standard *accumulate()* with a function object or function pointer as its fourth argument.

### 7.6.2 Inner Products

The function *inner\_product*() calculates the inner product or dot product of two sequences of floating point numbers. It is defined as

```
template<class In, class In2, class T>
T inner_product(In first, In last, In2 first2, T init) {
    while (first != last) init = init + *first++ * *first2++;
    return init;
}

template<class In, class In2, class T, class BinOp, class BinOp2>
T inner_product(In first, In last, In2 first2, T init,
               BinOp op, BinOp2, op2) {
    while (first != last)
        init = op(init, op2(*first++, *first2++));
    return init;
}
```

Note that only the beginning of the second sequence is passed as an argument and its size is implicitly assumed to be at least as long as the first sequence. Range checking is not provided here. The first version of *inner\_product*() does not apply to vectors of complex numbers since complex conjugation is needed for the second sequence. For inner products of complex vectors, the user may wish to define her own template instead of using the second version for efficiency reasons. For example,

```
void f(valarray<double>& v1, double* v2, vector<double>& v3){
    double d = inner_product(&v1[0], &v1[v1.size()], v2, 0.0);
    double e = inner_product(v3.begin(), v3.end(), &v1[0], 0.0);
}

template<class In, class In2, class T> // user-defined
T inner_product_cmpx(In first, In last, In2 first2, T init){
    while (first != last)
        init = init + *first++ * conj(*first2++);
    return init;
}

void g(valarray<complex<double>> &v1, complex<double>* v2){
    complex<double> d = 0;
    d = inner_product_cmpx(&v1[0], &v1[v1.size()], v2, d);
}
```

For a *vector* *vr* (§10.1.1), the functions *vr.begin()* and *vr.end()* point to the first and last-past-one elements of *vr*, respectively. However, a *valarray* *va* does not have functions *va.begin()* and *va.end()*. Fortunately, it forms a

sequence in the sense that  $va[0], va[1], \dots, va[va.size() - 1]$  are contiguous in memory, and thus standard algorithms can be applied.

### 7.6.3 Partial Sums

Given a sequence  $a_0, a_1, \dots, a_{n-1}$ , where  $n$  is a positive integer, the function template `partial_sum()` produces a sequence of partial sums, defined as  $a_0, a_0 + a_1, a_0 + a_1 + a_2, \dots, a_0 + a_1 + \dots + a_{n-1}$ . A general binary operation instead of summation can also be specified as an extra argument in the same way as in `accumulate()`. Its prototype is:

```
template<class In, class Out>          // summation by default
    Out partial_sum(In first, In last, Out res);

template<class In, class Out, class BinOp> // general op
    Out partial_sum(In first, In last, Out res, BinOp op);
```

It can be used as

```
void f(valarray<double>& v1, vector<double> v2) {
    partial_sum(&v1[0], &v1[v1.size()], &v1[0]);
    partial_sum(v2.begin(), v2.end(), v2.begin());
}
```

If  $v1$  has elements 1, 2, 5, 14, then the first call above changes it into the partial sum sequence 1, 3, 8, 22.

### 7.6.4 Adjacent Differences

Given a sequence  $a_0, a_1, \dots, a_{n-1}$ , where  $n$  is a positive integer, the function template `adjacent_difference()` produces a sequence of adjacent differences, defined as  $a_0, a_1 - a_0, a_2 - a_0 - a_1, \dots, a_{n-1} - a_0 - a_1 - \dots - a_{n-2}$ . A general binary operation instead of subtraction can also be specified as an extra argument in the same way as in `partial_sum()` and `accumulate()`. Its prototype is:

```
template<class In, class Out>          // subtract by default
    Out adjacent_difference(In first, In last, Out res);

template<class In, class Out, class BinOp> // general op
    Out adjacent_difference(In first, In last, Out res, BinOp op);
```

It can be used as

```
void f(valarray<double>& v1, vector<double> v2) {
    adjacent_difference(&v1[0], &v1[v1.size()], &v1[0]);
    adjacent_difference(v2.begin(), v2.end(), v2.begin());
}
```

If  $v1$  has elements 1, 3, 8, 22, then the first call above changes it into the adjacent difference sequence 1, 2, 5, 14. It is easy to see that the algorithms *adjacent\_difference()* and *partial\_sum()* are inverse operations.

## 7.7 Efficient Techniques for Numeric Integration

This section presents several efficient techniques for numeric integration. The techniques include using function objects instead of passing function pointers as arguments to function calls, using function pointers as template parameters, expression templates, and template metaprograms. These techniques are very general and also apply to many other situations. Two numeric integration methods, the Trapezoidal Rule and Simpson's Rule, are described in earlier sections. In §3.13 they are coded using traditional procedural programming style while in §5.1 a class is used for encapsulation. The programs in §3.13 and §5.1 work fine except for run-time efficiency, when they are called frequently. In this section, they are improved for run-time performance and generality by using templates.

### 7.7.1 Function Object Approach

As pointed out in §7.2 and §7.6, using templates and function objects can improve run-time efficiency for programs that traditionally require passing pointer-to-functions as arguments. Using a template and a function object, the function *trapezoidal()*, for numerically evaluating  $\int_a^b f(x)dx$ , can be written and used as

```
template<class Fo>
double trapezoidal(double a, double b, Fo f, int n) {
    double h = (b - a)/n;          // size of each subinterval
    double sum = f(a)*0.5;
    for (int i = 1; i < n; i++) sum += f(a + i*h);
    sum += f(b)*0.5;
    return sum*h;
}

class integrand {                // define a class for my integrand
public:
    double operator()(double x) { return exp(-x*x); }
};

int main() {
    cout << trapezoidal(0, 1, integrand(), 100) << '\n';
}
```



Recall that an object of a class with the function call operator `()` overloaded is called a *function object*. Thus the object `integrand()`, instead of the class name `integrand`, is passed to the function call `trapezoidal()`. The compiler will then instantiate `trapezoidal()`, substitute `integrand` for the template parameter `Fo`, which results in `integrand::operator()` being inlined into the definition of `trapezoidal()`. Note that `trapezoidal()` also accepts a pointer-to-function as its third argument, but this would be hard to be inlined and may not improve the run-time.

This technique can be put in a more general form that also takes a template parameter for the precision of integration:

```
template<class T> class Fo>
T trapezoidal(T a, T b, Fo f, int n) {
    T h = (b - a)/n;          // size of each subinterval
    T sum = f(a)*0.5;
    for (int i = 1; i < n; i++) sum += f(a + i*h);
    sum += f(b)*0.5;
    return sum*h;
}

template<class T> class integrand2 {
public:                        // define a class for my integrand
    T operator()(T x) { return exp(-x*x); }
};

int main() {
    cout << trapezoidal(0.0, 1.0, integrand2<double>(), 100);
}
```

With this form, integrals such as  $\int_0^1 e^{-x^2} dx$  can be integrated by the Trapezoidal Rule in *double*, *float*, *long double*, or other precisions.

### 7.7.2 Function Pointer as Template Parameter

The technique here to improve run-time efficiency still uses a function pointer, but in a very different way, in which the function pointer is used as a template parameter. Thus it can be inlined to avoid function call overheads. It can be illustrated by implementing the Trapezoidal Rule:

```
template<double F(double)> // F is a template parameter
double trapezoidal(double a, double b, int n) {
    double h = (b - a)/n;          // size of each subinterval
    double sum = F(a)*0.5;
    for (int i = 1; i < n; i++) sum += F(a + i*h);
    sum += F(b)*0.5;
    return sum*h;
}
```

```

}

double myintegrand(double x) {
    return exp(-x*x);
}

int main() {
    cout << trapezoidal<myintegrand>(0, 1, 100) << '\n';
}

```

With this approach, the integrand is still implemented as a function. But it is not passed as an argument to *trapezoidal*(). Instead, it is taken as a template parameter, which will be inlined. It can be generalized to

```

template<class T, T F(T)>
T trapezoidal(T a, T b, int n) {
    T h = (b - a)/n;          // size of each subinterval
    T sum = F(a)*0.5;
    for (int i = 1; i < n; i++) sum += F(a + i*h);
    sum += F(b)*0.5;
    return sum*h;
}

template<class T> T myintegrand2(T x) {
    return exp(-x*x);
}

int main() {
    cout << trapezoidal<double, myintegrand2>(0,1,100) << '\n';
}

```

### 7.7.3 Using Dot Products and Expression Templates

For integrals such as  $\int_a^b [f(x) + g(x)]dx$  and  $\int_a^b f(x)g(x)h(x)dx$ , the integrand is a product (or summation) of different functions such as  $f$ ,  $g$ , and  $h$ , and the C++ programs presented in previous sections can not be directly applied, since code such as

```

double result0 = trapezoidal(0, 1, f + g, 100);
double result2 = trapezoidal(0, 1, f*g*h, 100);

```

is illegal. One way is to write another function that returns the product or sum of some functions:

```

double F(double x, double f(double), double g(double)) {
    return f(x) + g(x);
}

```

```
double G(double x, double f(double), double g(double),
        double h(double)) {
    return f(x)*g(x)*h(x);
}
```

But the original integration function

```
template<class Fo>
double trapezoidal(double a, double b, Fo f, int n);
```

can no longer be applied to  $F$  or  $G$  (due to the complicated prototypes of  $F$  and  $G$ ), and thus several versions of it are needed. For a typical finite element analysis code, the integration of the product (or sum) of many functions (such as basis functions and their derivatives, coefficient functions and their derivatives) need be evaluated, and this approach would lead to many overloaded integration functions that are hard to manage and maintain.

In this subsection, dot products are used to evaluate numeric integration. As an example, consider a typical Gauss quadrature:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} w_i f(x_i), \quad (7.1)$$

where  $[x_0, x_1, \dots, x_{n-1}]$  are given Gauss points and  $[w_0, w_1, \dots, w_{n-1}]$  are the corresponding weights; see Exercise 5.10.10. If vector  $wgts$  contains the weights  $w_i$  and vector  $fxi$  contains the function values at Gauss points, then the dot product function can be called to give the approximate integral:

```
double dot(const Vtr& u, const Vtr& v) {
    double dotprod = 0;
    int n = u.size();
    for (int i = 0; i < n; i++) dotprod += u[i]*v[i];
    return dotprod;
}
```

```
Vtr fxi(n);
for (int i = 0; i < n; i++) fxi[i] = f(x[i]);
double d = dot(fxi, wgts); // dot() evaluates integrals
```

Here  $Vtr$  is a vector class (e.g., the standard library `vector<double>` or the class `Vcr` defined in Chapter 6). When an integrand is the product of many functions such as  $f * g * h$ , a vector-multiply function can be defined:

```
Vtr operator*(const Vtr& u, const Vtr& v) {
    int n = u.size();
    Vtr w(n);
    for (int i = 0; i < n; i++) w[i] = u[i]*v[i];
}
```

```
    return w;
}
```

```
Vtr fghxi = fxi*gxi*hxi;
double d0 = dot(fghxi, wgts); // evaluate integral of fgh
double d2 = dot(fxi, wgts);   // evaluate integral of f
```

where  $f_{xi}$ ,  $g_{xi}$ , and  $h_{xi}$  are the vectors consisting of the function values of  $f$ ,  $g$ , and  $h$ , respectively, evaluated at the Gauss points  $[x_0, x_1, \dots, x_{n-1}]$  of a given Gauss quadrature. This is a traditional way to evaluate integrals with integrands as products (or sums or a combination of them) of many different functions. It works fine and is very convenient to use, but performs very poorly due to the extra work in computing the loop for  $f_{xi} * g_{xi}$ , storing the result into a temporary vector  $tmp$ , computing another loop to obtain  $tmp * h_{xi}$ , and storing it in another temporary vector; see §6.5 for similar situations.

Expression templates can be applied to improve run-time efficiency of such vector multiplications by deferring intermediate operations and avoiding temporary objects. It is similar to the technique discussed in §6.5, but is more general. First define a class to hold left operand, operator, and right operand in an operation such as  $X * Y$ :

```
template<class LeftOpd, class Op, class RightOpd>
struct LOR {
    LeftOpd fod;        // store left operand
    RightOpd rod;       // store right operand

    LOR(LeftOpd p, RightOpd r): fod(p), rod(r) { }

    double operator[](int i) {
        return Op::apply(fod[i], rod[i]);
    }
};
```

Here the template parameter  $Op$  refers to a class whose member *apply()* defines a specific operation on elements of the left and right operands. It defers intermediate evaluations and thus avoids temporary objects and loops. Evaluation of the operation such as  $X * Y$  takes place only when the operator  $[]$  is called on an object of type *LOR*.

Then define a vector class, in which the operator  $=$  is overloaded that forces the evaluation of an expression:

```
class Vtr {
    int length;
    double* vr;
public:
    Vtr(int n, double* d) { length = n; vr = d; }
```

```

// assign an expression to a vector and
// do the evaluation
template<class LeftOpd, class Op, class RightOpd>
void operator=(LOR<LeftOpd, Op, RightOpd> exprn) {
    for (int i = 0; i < lenh; i++) vr[i] = exprn[i];
}

double operator[](int i) const { return vr[i]; }
};

```

Notice the function call *exprn[]* forces the operation *Op::apply()* to evaluate elements of the left and right operands.

Next define a class on how left and right operands are going to be evaluated:

```

// define multiply operation on elements of vectors
struct Multiply {
    static double apply(double a, double b) {
        return a*b;
    }
};

```

Finally overload the operator *\** for vector multiplication:

```

template<class LeftOpd>
LOR<LeftOpd, Multiply, Vtr> operator*(LeftOpd a, Vtr b) {
    return LOR<LeftOpd, Multiply, Vtr>(a,b);
}

```

The multiplication is actually deferred. Instead a small object of type *LOR* is returned. For example, *X \* Y* gives an object of *LOR* (without actually multiplying *X* and *Y*), where *Y* is of type *Vtr*, but *X* can be of a more general type such as *Vtr* and *LOR*.

They can now be used as

```

int main() {
    double a[] = { 1, 2, 3, 4, 5};
    double b[] = { 6, 7, 8, 9, 10};
    double c[] = { 11, 12, 13, 14, 15};
    double d[5];

    Vtr X(5, a), Y(5, b), Z(5, c), W(5, d);
    W = X * Y * Z;
}

```

When the compiler sees the statement  $W = X * Y * Z$ ; it constructs the object  $LOR<X, Multiply, Y>$  from  $X * Y$  and constructs the object

$$LOR< LOR<X, Multiply, Y>, Multiply, Z>$$

from  $LOR<X, Multiply, Y> * Z$  for the expression  $X * Y * Z$ . When it is assigned to object  $W$ , the operation  $Vtr::operator=$  is matched as

```
W.operator=(LOR< LOR<X, Multiply, Y>, Multiply, Z> exprn) {
    for (int i = 0; i < length; i++) vr[i] = exprn[i];
}
```

where  $exprn[i]$  is then expanded by inlining the  $LOR::operator[]$ :

```
Multiply::apply(LOR<Vtr,Multiply,Vtr>(X,Y)[i], Z[i]);
```

which is further expanded into

```
Multiply::apply(X[i], Y[i]) * Z[i]
= X[i] * Y[i] * Z[i].
```

The final result of  $W = X * Y * Z$  is:

```
for (int i = 0; i < length; i++) W[i] = X[i] * Y[i] * Z[i];
```

Thus there are no temporary vector objects or extra loops.

Similarly, vector multiplication  $V = W * X * Y * Z$  will be finally expanded by the compiler into

```
for (int i = 0; i < length; i++)
    V[i] = W[i] * X[i] * Y[i] * Z[i];
```

That is, the expression template technique efficiently computes the product of an arbitrary number of vectors with a single loop and without temporary vector objects.

For references on expression templates, see [Vel95, Fur97]. Some software libraries using expression templates to achieve run-time efficiency are PETE [Pet], POOMA [Poo], and Blitz++ [Bli]. Besides, the Web address <http://www.oonumerics.org> contains a lot of useful information on object-oriented numeric computing.

#### 7.7.4 Using Dot Products and Template Metaprograms

In §7.7.3, numeric integration is evaluated through dot products, where vector multiplication can be efficiently computed by using expression templates. A typical numeric integration formula (see Exercise 5.10.11 and §7.7.3) has very few terms in the summation, which implies that dot products of small vectors are usually evaluated. A dot product function is normally written using a loop:

```
double dot(const Vtr& u, const Vtr& v) {
    double dotprod = 0;
    int n = u.size();
    for (int i = 0; i < n; i++) dotprod += u[i]*v[i];
    return dotprod;
}
```

where the size of the vectors may not have to be known at compile-time.

For vectors of a small size (say 4), an alternative definition:

```
inline double dot4(const Vtr& u, const Vtr& v) {
    return u[0]*v[0] + u[1]*v[1] + u[2]*v[2] + u[3]*v[3];
}
```

can boost performance, since it removes loop overhead and low-level parallelism can be easily done on the summation operation. Besides, it is easier to make this small function to be inlined (reducing function-call overhead) and its data to be registerized. All these factors can make *dot4()* much more efficient than the more general *dot()* for vectors of size 4.

To improve performance, template metaprograms can be used so that the dot products of small vectors are expanded by the compiler in the form of *dot4()*. To illustrate the idea of template metaprograms, consider the Fibonacci numbers defined by a recursive relation:  $f_0 = 0, f_1 = 1, f_n = f_{n-1} + f_{n-2}$ , for  $n = 2, 3, \dots$ . They can be recursively computed by the compiler:

```
template<int N> struct fib{
    enum { value = fib<N-1>::value + fib<N-2>::value };
};

template<> struct fib<1>{
    enum { value = 1 };
};

template<> struct fib<0>{
    enum { value = 0 };
};

const int f3 = fib<3>::value;
const int f9 = fib<9>::value;
```

The compiler can generate that  $f_3 = 2$  and  $f_9 = 34$ . The recursive template behaves like a recursive function call (but at compile-time) and the specializations *fib<1>* and *fib<0>* stop the recursion.

This recursive template technique can be generalized to compute dot products. First define a class for small vectors:

```
template<int N, class T>
```

```

class smallVtr {
    T vr[N];          // array of size N and type T
public:
    T& operator[](int i) { return vr[i]; }
};

```

Then define a dot product function that calls a template metaprogram:

```

template<int N, class T>
inline T dot(smallVtr<N,T>& u, smallVtr<N,T>& v) {
    return metaDot<N>::f(u,v);
}

```

where the template metaprogram *metaDot* can be defined as

```

template<int M> struct metaDot {
    template<int N, class T>
    static T f(smallVtr<N,T>& u, smallVtr<N,T>& v) {
        return u[M-1]*v[M-1] + metaDot<M - 1>::f(u,v);
    }
};

template<> struct metaDot<1> {
    template<int N, class T>
    static T f(smallVtr<N,T>& u, smallVtr<N,T>& v) {
        return u[0]*v[0];
    }
};

```

The specialization *metaDot<1>* stops the recursion.

Now *dot* can be called on small vectors:

```

int main() {
    smallVtr<4, float> u, v;
    for (int i = 0; i < 4; i++) {
        u[i] = i + 1;
        v[i] = (i + 2)/3.0;
    }
    double d = dot(u,v);
    cout << " dot = " << d << '\n';
}

```

where the inlined function call *dot(u, v)* is expanded during compilation as

```

dot(u,v)
= metaDot<4>::f(u,v)
= u[3]*v[3] + metaDot<3>::f(u,v)
= u[3]*v[3] + u[2]*v[2] + metaDot<2>::f(u,v)
= u[3]*v[3] + u[2]*v[2] + u[1]*v[1] + metaDot<1>::f(u,v)

```



```
= u[3]*v[3] + u[2]*v[2] + u[1]*v[1] + u[0]*v[0]
```

It is this compile-time expanded code that gives better performance than a normal function call with a loop inside.

This template metaprogram technique could result in code that is several times faster than a normal function call when computing dot products of small vectors.

In large-scale applications, it is not enough to compute correctly, but also efficiently and elegantly. The elegance of a program should also include extendibility, maintainability, and readability, not just the look of the program. The code for dot products of small vectors using the template metaprogram technique is also elegant in the sense that it can be easily extended to other situations, for example, dot products of complex vectors, and can be maintained easily. Writing it in FORTRAN or C styles would possibly result in tens of versions (dot products of vectors of sizes 2, 3, 4, 5, and so on, and for real and complex small vectors in different precisions). The Trapezoidal Rule itself is also written in efficient and elegant ways in this section and many other functions such as Simpson's Rule can be as well; see Exercise 7.9.11.

## 7.8 Polynomial Interpolation

Given a set of  $n+1$  values  $\{y_i\}_{i=0}^n$  of a function  $f(x)$  at  $n+1$  distinct points  $\{x_i\}_{i=0}^n$ , where  $n$  is a positive integer, how can one approximate  $f(x)$  for  $x \neq x_i$ ,  $i = 0, 1, \dots, n$ ? Such a problem arises in many applications in which the  $n+1$  pairs  $\{x_i, y_i\}$  may be obtained from direct measurement, experimental observations, or an expensive computation, but function values  $f(x)$  at other points are needed. This section deals with *polynomial interpolation*, in which a polynomial  $p_n(x)$  of degree  $n$  is constructed to approximate  $f(x)$  that satisfies the conditions  $f(x_i) = p_n(x_i)$ ,  $i = 0, 1, \dots, n$ .

### 7.8.1 Lagrange Form

The approximating polynomial  $p_n(x)$  can be written in the Lagrange form:

$$p_n(x) = y_0 L_0^{(n)}(x) + y_1 L_1^{(n)}(x) + \cdots + y_n L_n^{(n)}(x) = \sum_{i=0}^n y_i L_i^{(n)}(x),$$

where each  $L_i^{(n)}(x)$  is a polynomial of degree  $n$  and depends only on the interpolation nodes  $\{x_i\}_{i=0}^n$ , but not on the given function values  $\{y_i\}_{i=0}^n$ . They are defined as

$$L_i^{(n)}(x) = \prod_{\substack{j=0 \\ j \neq i}}^n \frac{x - x_j}{x_i - x_j}, \quad i = 0, 1, \dots, n-1.$$

That is, each  $L_i^{(n)}(x)$  is the product of all factors  $(x - x_j)/(x_i - x_j)$  for  $j \neq i$ . They have the following property.

$$L_i^{(n)}(x_j) = 0 \quad \text{for } j \neq i, \quad \text{and } L_i^{(n)}(x_i) = 1.$$

Thus  $p_n(x)$  agrees with  $f(x)$  at the given interpolation nodes  $\{x_i\}_{i=0}^n$ . Function value  $f(x)$  for  $x$  other than the interpolation nodes can be approximated by  $p_n(x)$ .

When  $n = 1$ , the interpolation polynomial  $p_1(x)$  is a line passing through  $(x_0, y_0)$  and  $(x_1, y_1)$ , and has the form:

$$p_1(x) = L_0^{(1)}(x)y_0 + L_1^{(1)}(x)y_1 = \frac{x - x_1}{x_0 - x_1}y_0 + \frac{x - x_0}{x_1 - x_0}y_1.$$

This is the so-called linear interpolation, which is used frequently in many scientific works. When  $n = 2$ ,  $p_2(x)$  is a quadratic polynomial passing through  $(x_0, y_0)$ ,  $(x_1, y_1)$ , and  $(x_2, y_2)$ , and has the form:

$$\begin{aligned} p_2(x) &= \sum_{i=0}^2 L_i^{(2)}(x)y_i \\ &= \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)}y_0 + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)}y_1 + \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}y_2. \end{aligned}$$

Since some applications require more accuracy than others, a template function can be written so that a user can conveniently choose single, double, long double, or other precisions:

```
template<class T>
T lagrange(const vector<T>& vx, const vector<T>& vy, T x) {
    int n = vx.size() - 1;
    T y = 0;
    for (int i = 0; i <= n; i++) {
        T temp = 1;
        for (int j = 0; j <= n; j++)
            if (j != i) temp *= (x - vx[j]) / (vx[i] - vx[j]);
        y += temp*vy[i];
    }
    return y;
}
```

The vectors  $vx$  and  $vy$  store, respectively, the given  $x$  and  $y$  coordinates of the interpolation points. The function template computes the approximate function value at  $x$ . In particular, the inner *for* loop calculates  $L_i^{(n)}(x)$  and the outer *for* loop gives the summation  $\sum_{i=0}^n y_i L_i^{(n)}(x)$ .

As an example, consider the interpolation problem that gives  $x_i = 1 + i/4.0$  and  $y_i = e^{x_i}$  for  $i = 0, 1, 2, 3$ . The function template can then be used to find an approximate function value at  $x = 1.4$ :

```

int main() {
    const int n = 4;
    vector<float> px(n);
    vector<float> py(n);
    for (int i = 0; i < n; i++) {
        px[i] = 1 + i/4.0; py[i] = exp( px[i] );
    }
    float x = 1.4;
    float approximation = lagrange(px, py, x);
}

```

Compared to the exact function value  $e^{1.4}$ , the error of this approximation is 0.0003497 (on my machine).

### 7.8.2 Newton Form

The Newton form is more efficient and seeks  $p_n(x)$  as

$$p_n(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + \cdots + c_n(x - x_0) \cdots (x - x_{n-1}),$$

where  $c_0, c_1, \dots, c_n$  are constants to be determined. For example, the linear and quadratic interpolation polynomials in Newton's form are

$$\begin{aligned} p_1(x) &= c_0 + c_1(x - x_0), \\ p_2(x) &= c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1). \end{aligned}$$

Before presenting an algorithm on how to find  $c_i$ ,  $i = 0, 1, \dots, n$ , we talk about how Newton's form can be evaluated efficiently, assuming all the  $c_i$  are known. Write  $p_n(x)$  into a nested multiplication form:

$$p_n(x) = c_0 + d_0(c_1 + d_1(c_2 + \cdots + d_{n-3}(c_{n-2} + d_{n-2}(c_{n-1} + d_{n-1}(c_n))) \cdots)),$$

where  $d_0 = x - x_0$ ,  $d_1 = x - x_1, \dots, d_{n-1} = x - x_{n-1}$ , and introduce the notation:

$$\begin{aligned} u_n &= c_n; \\ u_{n-1} &= c_{n-1} + d_{n-1}u_n; \\ u_{n-2} &= c_{n-2} + d_{n-2}u_{n-1}; \\ &\vdots \\ u_1 &= c_1 + d_1u_2; \\ &\vdots \\ u &\leftarrow c_n; \end{aligned}$$

Then  $p_n(x) = u_0$ . This gives the following algorithm:

$$u \leftarrow c_n;$$

```

for ( $i = n - 1, n - 2, \dots, 0$ ) {
     $u \leftarrow c_i + d_i * u;$            // that is:  $u \leftarrow c_i + (x - x_i) * u$ 
}
return  $u;$ 

```

This is the so-called Horner's algorithm (see §3.12), which is very efficient in evaluating polynomials.

Below is a procedure on how to find the coefficients  $c_i$ . The requirement  $p_n(x_i) = f(x_i)$  leads to

$$\begin{aligned}
 f(x_0) &= p_n(x_0) = c_0, \\
 f(x_1) &= p_n(x_1) = c_0 + c_1(x_1 - x_0), \\
 f(x_2) &= p_n(x_2) = c_0 + c_1(x_2 - x_0) + c_2(x_2 - x_0)(x_2 - x_1), \\
 &\vdots
 \end{aligned}$$

Thus  $c_0 = f(x_0)$  and  $c_1 = (f(x_1) - f(x_0))/(x_1 - x_0)$ . Define the divided differences of order 0 as

$$f[x_i] = f(x_i), \quad i = 0, 1, \dots, n,$$

and of order 1 as

$$f[x_i, x_j] = \frac{f[x_j] - f[x_i]}{x_j - x_i}, \quad i, j = 0, 1, \dots, n, \text{ and } i \neq j,$$

and of order  $j$  as

$$\begin{aligned}
 &f[x_i, x_{i+1}, \dots, x_{i+j}] \\
 &= \frac{f[x_{i+1}, x_{i+2}, \dots, x_{i+j}] - f[x_i, x_{i+1}, \dots, x_{i+j-1}]}{x_{i+j} - x_i}, \quad i, j = 0, 1, \dots, n.
 \end{aligned}$$

Then  $c_0 = f[x_0]$ ,  $c_1 = f[x_0, x_1]$ , and

$$\begin{aligned}
 c_2 &= \frac{f(x_2) - f(x_0) - c_1(x_2 - x_0)}{(x_2 - x_0)(x_2 - x_1)} \\
 &= \frac{\frac{f(x_2) - f(x_1)}{x_2 - x_1} + \frac{f(x_1) - f(x_0) - c_1(x_2 - x_0)}{x_2 - x_1}}{x_2 - x_0} \\
 &= \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0} \\
 &= f[x_0, x_1, x_2].
 \end{aligned}$$

Similarly, it can be shown that  $c_i = f[x_0, x_1, \dots, x_i]$  for  $i = 0, 1, \dots, n$ . That is, all the coefficients  $c_i$  can be expressed as divided differences and

$x_0$	$f[x_0]$	$f[x_0, x_1]$	$f[x_0, x_1, x_2]$	$f[x_0, x_1, x_2, x_3]$
$x_1$	$f[x_1]$	$f[x_1, x_2]$	$f[x_1, x_2, x_3]$	
$x_2$	$f[x_2]$	$f[x_2, x_3]$		
$x_3$	$f[x_3]$			

TABLE 7.1. A table of divided differences in the case of  $n = 3$ .

the interpolation polynomial  $p_n(x)$  is:

$$p_n(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + \cdots + f[x_0, x_1, \dots, x_n](x - x_0)(x - x_1) \cdots (x - x_{n-1}).$$

If a table of function values  $(x_i, f(x_i))$  is given, then a table of divided differences can be constructed recursively column by column as indicated in Table 7.1 for the case of  $n = 3$ . In particular, the first two columns list the given  $x$  and  $y$  coordinates of the interpolation points, divided differences of order 1 are computed and put in the third column, divided differences of order 2 are computed and put in the fourth column, and so on. When the table is completed, the divided differences in the top row will be used to compute  $p_n(x)$ .

For example, given a table of interpolation nodes and corresponding function values:

$x$	3	1	5	6
$f(x)$	1	-3	2	4

the table of divided differences can be computed according to Table 7.1 as

3	1	2	-3/8	7/40
1	-3	5/4	3/20	
5	2	2		
6	4			

Then the interpolation polynomial can be written as

$$p_3(x) = 1 + 2(x - 3) - \frac{3}{8}(x - 3)(x - 1) + \frac{7}{40}(x - 3)(x - 1)(x - 5).$$

When  $n$  is large, calculating the divided difference table can not be done by hand. To derive an efficient algorithm for doing this, introduce the notation  $c_{i,j} = f[x_i, x_{i+1}, \dots, x_{i+j}]$ . Then Table 7.1 of divided differences becomes (in the case of  $n = 4$ ):

$x_0$	$c_{0,0}$	$c_{0,1}$	$c_{0,2}$	$c_{0,3}$	$c_{0,4}$
$x_1$	$c_{1,0}$	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	
$x_2$	$c_{2,0}$	$c_{2,1}$	$c_{2,2}$		
$x_3$	$c_{3,0}$	$c_{3,1}$			
$x_4$	$c_{4,0}$				

This actually leads to the following algorithm:

```

for ( $k = 0, 1, \dots, n$ ) {
     $c_{k,0} \leftarrow f(x_k)$ ;
}
for ( $j = 1, 2, \dots, n$ ) {
    for ( $i = 0, 1, \dots, n - j$ ) {
         $c_{i,j} \leftarrow (c_{i+1,j-1} - c_{i,j-1}) / (x_{i+j} - x_i)$ ;
    }
}

```

Then  $p_n(x)$  can be obtained as

$$p_n(x) = c_{0,0} + c_{0,1}(x - x_0) + c_{0,2}(x - x_0)(x - x_1) + \dots + c_{0,n}(x - x_0)(x - x_1) \cdots (x - x_{n-1}).$$

A closer look at the algorithm reveals that only  $c_{0,0}, c_{0,1}, \dots, c_{0,n}$  are needed in constructing  $p_n(x)$ , and other  $c_{i,j}$  for  $i \neq 0$  are just intermediate results. A space-efficient version of the algorithm requires only an array  $b = [b_0, b_1, \dots, b_n]$  of size  $n + 1$ . First store the given function value  $f(x_i)$  into  $b_i$ , for  $i = 0, 1, \dots, n$ . Note that  $b_0 = c_{0,0}$ . Then compute divided differences  $c_{i,1}$  for  $i = 0, 1, \dots, n - 1$  and store them in  $b_1, b_2, \dots, b_n$ . Now  $b_0 = c_{0,0}$  and  $b_1 = c_{0,1}$  are the first two desired coefficients for  $p_n(x)$ . Next compute divided differences  $c_{i,2}$  for  $i = 0, 1, \dots, n - 2$  and store them in  $b_2, b_3, \dots, b_n$ . Now  $b_0 = c_{0,0}$ ,  $b_1 = c_{0,1}$ , and  $b_2 = c_{0,2}$ . Continue this process until  $c_{0,n}$  is computed and stored in  $b_n$ . Now the revised and more space-efficient algorithm for calculating the coefficients of  $p_n(x)$  reads:

```

for ( $k = 0, 1, \dots, n$ ) {
     $b_k \leftarrow f(x_k)$ ;
}
for ( $j = 1, 2, \dots, n$ ) {
    for ( $i = n, n - 1, \dots, j$ ) {
         $b_i \leftarrow (b_i - b_{i-1}) / (x_i - x_{i-j})$ ;
    }
}

```

After this algorithm computes all  $b_k$ , the interpolation polynomial  $p_n(x)$  can be finally constructed in Newton's form:

$$p_n(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1) + \cdots + b_n(x - x_0)(x - x_1) \cdots (x - x_{n-1}).$$

Combining this algorithm and the one for nested multiplication leads to the following code for computing Newton's form of the interpolation polynomial  $p_n(x)$ .

```
template<class T>
T newton(const vector<T>& vx, const vector<T>& vy, T x) {
    vector<T> b = vy;
    int n = vx.size() - 1;

    // find coefficients in Newton's form
    for (int j = 1; j <= n; j++)
        for (int i = n; i >= j; i--)
            b[i] = (b[i] - b[i-1])/(vx[i] - vx[i-j]);

    // evaluate interpolation polynomial at x
    T u = b[n];
    for (int i = n - 1; i >= 0; i--) u = b[i] + (x - vx[i])*u;
    return u;
}
```

Applying this program to the example presented at the end of §7.8.1 produces about the same result as *lagrange()*.

The advantage of Newton's form is that it is very efficient and the coefficients  $c_i$ ,  $i = 0, 1, \dots, n$ , can be used when later there are more interpolation points available. For example, if later one more point  $(x_{n+1}, f(x_{n+1}))$  is given, then  $p_{n+1}(x) = p_n(x) + c_{n+1}(x - x_0) \cdots (x - x_n)$  gives  $c_{n+1} = (f(x_{n+1}) - p_n(x_{n+1})) / ((x_{n+1} - x_0) \cdots (x_{n+1} - x_n))$ .

## 7.9 Exercises

- 7.9.1. Test the class template for vectors *Vcr<T>* presented in §7.1 and add more operations such as vector multiplication and addition, and 2-norm (see §6.3).
- 7.9.2. Test the class template for vectors *Vcr<T>* together with its specializations *Vcr<complex<T>>* and *Vcr<complex<double>>* presented in §7.1. Also add more operations such as vector multiplication and addition, and 2-norm (see §6.3).

- 7.9.3. Turn the matrix class *Mtx* defined in §6.3 into a class template.
- 7.9.4. Traits can be used as an alternative to certain template specializations or function overloading. For example, the *maxnorm()* function for arrays of real or complex numbers can be defined as

```
template<class T> struct RTrait {
    typedef T RType; // for return type of maxnorm()
};

template<class T> struct RTrait< complex<T> > {
    typedef T RType; // return type for complex arrays
}; // a specialization of RTrait<T>

template<class T> // definition of maxnorm
typename RTrait<T>::RType maxnorm(T u[], int n) {
    typename RTrait<T>::RType nm = 0;
    for (int i = 0; i < n; i++) nm = max(nm, abs(u[i]));
    return nm;
}
```

Here the trait class *RTrait* defines the return type for the function *maxnorm()*; but for complex vectors, the return value of a norm function must be a real number. The keyword *typename* must be used to instruct the compiler that *RTrait<T>::RType* is a type name. One definition of the function handles real and complex vectors. Rewrite the vector class *Vcr<T>* in §7.1 using this approach for the member function *maxnorm()*. This approach would be much better if *maxnorm()* were a large function. Can it be applied to vector dot products?

- 7.9.5. Apply the sort function template *sort<T>()* defined in §7.2 to sort a vector of complex numbers in decreasing order according to their absolute values, by overloading the operator < for complex numbers.
- 7.9.6. Apply the sort function template *sort<T>()* defined in §7.2.3 to sort a vector of complex numbers in decreasing order according to their absolute values, by using a specialization for the comparison template *less<T>()*.
- 7.9.7. Write a specialization for the template *less<T>()* so that the sort function template *sort<T>()* in §7.2.3 will sort a vector of *point2d* in decreasing order according to the *y*-coordinate.
- 7.9.8. Apply function template *compare<T>()*, defined in §7.2.4, to compare two vectors of *char* with case-sensitive and case-insensitive comparisons, respectively.



- 7.9.9. Write a function object to be used as an argument of the function template *accumulate()*, presented in §7.6, for calculating the 2-norm of a vector of complex numbers. Write a function template, similar to *accumulate()*, that calculates directly the 2-norm of a vector of complex numbers. Compare the efficiency of the two function templates.
- 7.9.10. Write the deferred-evaluation operator overloading for the vector *saxpy* operation as presented in §6.5 into a template so that it can handle vectors of single, double, and long double precisions.
- 7.9.11. Write Simpson's Rule for numeric integration into the more efficient and elegant forms as described in §7.7.
- 7.9.12. Write a template metaprogram to compute the factorial of small integers (e.g., less than 12; see Exercise 1.6.2) during compilation.
- 7.9.13. Use expression templates to write a program to add an arbitrary number of vectors without introducing temporary vector objects or extra loops.
- 7.9.14. Write the conjugate gradient algorithm (§6.6) into a member function of a class template so that it can be used to solve real linear systems  $Ax = b$  for symmetric and positive definite matrix  $A$  and vectors  $b$  and  $x$  in single, double, and long double precisions.
- 7.9.15. Write the conjugate gradient algorithm (§6.6) into a member function of a class template with specializations for vector dot products so that it can be used to solve real and complex linear systems  $Ax = b$  for Hermitian and positive definite matrix  $A$  and vectors  $b$  and  $x$  in single, double, and long double precisions for real and imaginary parts.
- 7.9.16. Given a set of values of function  $f(x)$ :

$x$	5	7	6	6.6
$f(x)$	1	-23	-50	-4

apply Lagrange and Newton forms of the interpolation polynomial to approximate  $f(x)$  for  $x = 5.5$  and  $x = 6.2$ .



# 8

## Class Inheritance

A new class can be derived from an existing class. The new class, called *derived class*, then inherits all members of the existing class, called *base class*, and may provide additional data, functions, or other members. This relationship is called *inheritance*. The derived class can then behave as a subtype of the base class and an object of the derived class can be assigned to a variable of the base class through pointers and references. This is the essential part of what is commonly called object-oriented programming. This chapter deals with various issues in class inheritance. The disadvantage of object-oriented programming is that it may sometimes greatly affect run-time performance. The last section of the chapter presents techniques on what can be done when performance is a big concern.

### 8.1 Derived Classes

Suppose we want to define classes for points in one, two, and three dimensions. A point in 1D can be defined as

```

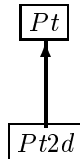
class Pt {                                // class for 1D point
private:
    double x;                             // x coordinate
public:
    Pt(double a = 0) { x = a; }           // constructor
    void draw() const { cout << x; }      // draw the point
};

```

The function `draw()` is defined to simply print out the coordinate. Since a 2D point has both  $x$  and  $y$  coordinates, it can be defined to inherit the  $x$  coordinate from the class `Pt` and provide a new member for the  $y$  coordinate:

```
class Pt2d: public Pt {                // class for 2D point
private:                               // inherits x from Pt
    double y;                          // y coordinate
public:
    Pt2d(double a = 0, double b = 0): Pt(a), y(b) { }
    void draw() const {                // draw x coordinate
        Pt::draw();                  // by Pt's draw()
        cout << " " << y;            // draw y coordinate
    }
};
```

By putting `:public Pt` after the name of the class `Pt2d` when it is declared, the new class `Pt2d` is defined to be a *derived class* from the *base class* `Pt`. The derived class `Pt2d` is also called a *subclass* and the base class `Pt` a *superclass*. `Pt2d` has members of class `Pt` in addition to its own members. The derived class `Pt2d` is often said to inherit properties from its base class `Pt`. This relationship is called *inheritance*. It can be represented graphically by an arrow from the derived class to the base:



Now a `Pt2d` is also a `Pt` (a two-dimensional point can be regarded as a one-dimensional point by just looking at the  $x$  coordinate), and `Pt2d*` (pointer to `Pt2d`) can be used as `Pt*`. However, a `Pt` is not necessarily a `Pt2d` and a `Pt*` can not be used as a `Pt2d*`. The keyword `public` in the definition of `Pt2d` means that the derived class `Pt2d` has a public base `Pt`, which in turn means that a `Pt2d*` can be assigned to a variable of type `Pt*` by any function without an explicit type conversion. (Inheritance using *private* and *protected* bases is discussed in §8.3.) The opposite conversion, from a pointer to base class to a pointer to derived class, must be explicit. For example,

```
void f(Pt p1, Pt2d p2) {
    Pt* q1 = &p2;                // OK. Every Pt2d is a Pt
    Pt2d* q2 = &p1;              // error, not every Pt is a Pt2d
}
```

```

void g(Pt p1, Pt2d p2) {      // statements below are legal
    Pt* q1 = &p2;             // Every Pt2d is a Pt
    Pt2d* q2 = static_cast<Pt2d*>(q1);
                                // explicit type conversion

    q2->draw();                // x, y of p2 are printed

    Pt2d* r2 = static_cast<Pt2d*>(&p1);
                                // explicit type conversion

    r2->draw();                // OK, but y may be garbage
}

int main() {                  // test what is printed out

    Pt a(5);                  // create a 1D point
    Pt2d b(4, 9);             // create a 2D point
    g(a, b);                  // call g() on a and b

}

```

During the second draw in calling *g()*, the 1D point *a* does not have a *y*-coordinate and thus some garbage should be printed out. The first draw in *g()* on the 2D point *b* should print out both coordinates of *b* correctly.

An object of a derived class can be treated as an object of its base class when manipulated through pointers and references. The opposite must be through explicit type conversion using *static\_cast* (at compile-time) or *dynamic\_cast* (at run-time; see §8.5), although the result of such conversions can not be guaranteed in general. For example, the second cast in the function *g()* above tries to convert a pointer to a base class to a pointer to a derived class and may result in an object with an undefined *y* coordinate. The operator *static\_cast* converts between related types such as from one pointer type to another or from an enumeration to an integral type. There is another cast, called *reinterpret\_cast*, which converts between unrelated types such as an integer to a pointer. These casts may not be portable, can be even dangerous, but are sometimes necessary. They should be avoided when possible.

Note that the function *draw()* may draw garbage for the *y* coordinate in *r2->draw()* above. A closer look reveals that *r2* is cast from a one-dimensional point. It would be nice if the system could detect the type of an object and invoke the draw function *Pt::draw()* for one-dimensional points and invoke *Pt2d::draw()* for two-dimensional points. In §8.1.5, virtual functions are introduced so that for a 1D point, the function *Pt::draw()* will be invoked and for a 2D point, *Pt2d::draw()* will be invoked automatically.

### 8.1.1 Member Functions

Member functions of a derived class can not access the private part of a base class, although a derived class contains all members of the base class. For example, the function `Pt2d::draw()` can not be defined as

```
void Pt2d::draw() const { // incorrect
    cout << x << " " << y; // x is not accessible
}
```

A member of a derived class that is inherited from the private part of a base class may be called an invisible member of the derived class. Thus an invisible member of a class can not be accessed by members or friends even of the same class. This is for information hiding, since otherwise a private member of a class could be accessed freely just by deriving a new class from it.

However, the hidden member `x` of the derived class `Pt2d` is accessible through the public member function `Pt::draw()` :

```
void Pt2d::draw() const { // draw the point
    Pt::draw();           // draw x coordinate by Pt's draw()
    cout << " " << y;      // draw y coordinate
}
```

Note the class qualifier `Pt::` must be used to refer to a member function of class `Pt`.

### 8.1.2 Constructors and Destructors

In defining the constructor of a derived class, the constructor of the base class must be called in the initializer list (like class object members; see §5.6) if the base class has a constructor. For example,

```
Pt2d::Pt2d(double a = 0, double b = 0): Pt(a), y(b) { }
```

It can also be defined alternatively as

```
Pt2d::Pt2d(double a = 0, double b = 0): Pt(a) { y = b; }
```

But it can not be defined as

```
Pt2d::Pt2d(double a = 0, double b = 0) { Pt(a); y = b; }
```

The third definition is wrong since the construction of base class `Pt` is not put in the initializer list.

A derived class constructor can specify initializers for its own members and immediate bases only. The construction of an object of a derived class starts from the base class, then the members, and then the derived class itself. Its destruction is in the opposite order: first the derived class itself,

then its members, and then the base class. Members and bases are constructed in order of declarations in the derived class and destructed in the reverse order.

In the definition of the destructor of a derived class, only spaces allocated in the derived class need be explicitly deallocated; spaces allocated in the base class are freed implicitly by the destructor of the base class.

### 8.1.3 Copying

Copying of class objects is defined by the copy constructor and assignment. For example,

```
class Pt {
public:          // ... in addition to other members
    Pt(const Pt& p) { x = p.x; }    // copy constructor
    Pt& operator=(const Pt& p) {    // copy assignment
        if (this != &p) x = p.x;
        return *this;
    }
};

Pt2d p2;
Pt p1 = p2;          // construct p1 from Pt part of p2
p1 = p2;             // assign Pt part of p2 to p1
```

Since the copy functions of *Pt* do not know anything about *Pt2d*, only the *Pt* part (*x*-coordinate) of a *Pt2d* object is copied and other parts are lost. This is commonly called slicing. It can be avoided by passing pointers and references; see §8.1.5.

### 8.1.4 Class Hierarchy

A derived class can be a base class of another derived class. For example,

```
class Pt3d: public Pt2d {    // point in 3D
private:                   // inherits x,y from Pt2d
    double z;              // z coordinate
public:
    Pt3d(double a = 0, double b = 0, double c = 0)
        : Pt2d(a, b), z(c) { }
    void draw() const {    // draw the point
        Pt2d::draw();     // draw x,y by Pt2d's draw()
        cout << " " << z; // draw z coordinate
    }
};
```

A set of related classes derived through inheritance is called a *class hierarchy*.

A derived class can inherit from two or more base classes. This is called *multiple inheritance*; see §8.4.

### 8.1.5 Virtual Functions

Virtual functions can be declared in a base class and may be redefined (also called overridden) in each derived class when necessary. They have the same name and same set of argument types in both base class and derived class, but perform different operations. When they are called, the system can guarantee the correct function be invoked according to the type of the object at run-time. For example, the class *Pt* can be redefined to contain a virtual function:

```
class Pt {
private:
    double x;
public:
    Pt(double a = 0) { x = a; }
    virtual void draw() const { cout << x; }    // virtual fcn
};
```

The member *Pt::draw()* is declared to be a virtual function. The declarations of *Pt2d* and *Pt3d* remain unchanged. The keyword *virtual* indicates that *draw()* can act as an interface to the *draw()* function defined in this class and the *draw()* functions defined in its derived classes *Pt2d* and *Pt3d*. When *draw()* is called, the compiler will generate code that chooses the right *draw()* for a given *Pt* object (*Pt2d* and *Pt3d* objects are also *Pt* objects). For example, if a *Pt* object is actually a *Pt3d* object, the function *Pt3d::draw()* will be called automatically by the system.

A virtual function can be used even if no class is derived from the class, and a derived class need not redefine it if the base class version of the virtual function works fine. Now functions can be defined to print out a set of points (some may be 1D or 2D while others may be 3D):

```
void h(const vector<Pt*>& v) {                // include <vector>
    for (int i = 0; i < v.size(); i++) {
        v[i]->draw();
        cout << '\n';
    }
}

int main() {                                // test what is printed out
    Pt a(5);
    Pt2d b(4, 9);
```



```

Pt3d c(7, 7, 7);
vector<Pt*> v(3);
v[0] = &a;
v[1] = &b;
v[2] = &c;
h(v);
}

```

For library `<vector>` see §10.1.1. This will work even if the function `h()` was written and compiled before the derived classes `Pt2d` and `Pt3d` were even conceived of, and the points (1D, 2D, or 3D) in the argument of `h()` can be generated dynamically at run-time by another function. Since `draw()` is declared to be *virtual* in the base class, the system guarantees that in the call `v[i]—>draw()`, the function `Pt::draw()` is invoked if `v[i]` is a pointer to `Pt`, `Pt2d::draw()` is invoked if `v[i]` is a pointer to `Pt2d`, and `Pt3d::draw()` is invoked if `v[i]` is a pointer to `Pt3d`. This is a key aspect of class inheritance. When used properly, it is a cornerstone of object-oriented design.

Getting the right correspondence between the function (e.g., among different versions of `draw()`) and the type (e.g., among `Pt`, `Pt2d`, or `Pt3d`) of an object is called run-time *polymorphism*. A type with virtual functions is called a *polymorphic type*. Run-time polymorphism is also called *dynamic binding*. To get polymorphic behavior, member functions called must be *virtual* and objects must be manipulated through pointers or references. When manipulating objects directly (rather than through pointers or references), their exact types must be known at compile-time, for which run-time polymorphism is not needed and templates may be used. See §8.6 for examples that can be implemented using either templates or virtual functions. In contrast, what templates provide is often called compile-time polymorphism, or *static polymorphism*.

When run-time polymorphism is not needed, the scope resolution operator `::` should be used, as in `Pt::draw()` and `Pt2d::draw()`. When a virtual function is inline, function calls that do not need run-time polymorphism can be specified by using the scope resolution operator `::` so that inline substitution may be made. This is reflected in the definitions of `Pt2d::draw()` and `Pt3d::draw()`.

Pointers to class members were discussed in §5.8. A function invoked through a pointer to a member function can be a *virtual* function, and polymorphic behavior can be achieved through pointers to virtual member functions.

### 8.1.6 Virtual Destructors

A virtual destructor is a destructor that is also a virtual function. Proper cleanup of data can be ensured by defining a virtual destructor in a base

class and overriding it in derived classes. For example, consider the following inheritance without a virtual destructor.

```
class B {
    double* pd;
public:
    B() {                                // constructor of B
        pd = new double [20];
        cout << "20 doubles allocated\n";
    }
    ~B() {                               // destructor of B
        delete[] pd;
        cout << "20 doubles deleted\n";
    }
};

class D: public B {                     // derive D from B
    int* pi;
public:
    D(): B() {                          // constructor of D
        pi = new int [1000];
        cout << "1000 ints allocated\n";
    }
    ~D() {                              // destructor of D
        delete[] pi;
        cout << "1000 ints deleted\n";
    }
};
```

Then the code

```
int main() {
    B* p = new D;
        // 'new' constructs a D object
    delete p;
        // 'delete' frees a B object since p is a pointer to B
}
```

will produce the output

```
20 doubles allocated
1000 ints allocated
20 doubles deleted
```

Proper cleanup is not achieved here. The reason is that the *new* operator constructed an object of type *D* (allocated 20 doubles and 1000 ints when *D*'s constructor was called), but the *delete* operator cleaned up an object of type *B* pointed to by *p* (freed 20 doubles when *B*'s destructor was implicitly

called). In other words, run-time polymorphism did not apply to the destructor of class *D*. If it were applied, the system would have detected the type of the object pointed to by *p* and called the destructor of the correct type (class *D*).

This problem can be easily solved by declaring the destructor of the base class *B* to be *virtual* :

```
class B {
    double* pd;
public:
    B() {
        pd = new double [20];
        cout << "20 doubles allocated\n";
    }
    virtual ~B() {                // a virtual destructor
        delete[] pd;
        cout << "20 doubles deleted\n";
    }
};
```

The definition of the derived class *D* is unchanged. Now the main program above produces the following desired output.

```
20 doubles allocated
1000 ints allocated
1000 ints deleted
20 doubles deleted
```

From this, the orders of construction and destruction of objects of derived and base classes can also be clearly seen; see §8.1.2.

In general, a class with a virtual function should have a virtual destructor, because run-time polymorphism is expected for such a class. However, other base classes such as class *B* above may also need a virtual destructor for proper cleanups.

## 8.2 Abstract Classes

Some virtual functions in a base class can only be declared but can not be defined, since the base class may not have enough information to do so and such virtual functions are only meant to provide a common interface for the derived classes. A virtual function is called a *pure virtual function* if it is declared but its definition is not provided. The initializer “= 0” makes a virtual function a pure one. For example,

```
class Shape {
public:
```

```

    virtual void draw() = 0;          // pure virtual function
    virtual void rotate(int i) = 0;  // rotate i degrees
    virtual bool is_closed() = 0;    // pure virtual function
    virtual ~Shape() { }             // empty virtual destructor
};

```

The member functions *rotate()*, *draw()*, and *is\_closed()* are pure virtual functions recognized by the initializer “= 0”. Note the destructor *~Shape()* is not a pure virtual function since it is defined, although it does nothing. This is an example of a function that does nothing but is useful and necessary. The class *Shape* is meant to be a base class from which other classes such as *Triangle* and *Circle* can be derived. Its member functions *rotate()*, *draw()*, and *is\_closed()* can not be defined since there is not enough information to do so.

A class with one or more pure virtual functions is called an *abstract class*. No objects of an abstract class can be created. For example, it is illegal to define:

```
Shape s;    // object of an abstract class can not be defined
```

An abstract class can only be used as an interface and as a base for derived classes. For example, a special shape: *Circle* can be derived from it as

```

class Circle: public Shape {    // derived class for circles
private:
    Pt2d center;                // center of circle
    double radius;              // radius of circle
public:
    Circle(Pt2d, double);        // constructor
    void rotate(int) { }         // override Shape::rotate()
    void draw();                 // override Shape::draw()
    bool is_closed() { return true; } // a circle is closed
};

```

The member functions of *Circle* can be defined since there is enough information to do so. These functions override the corresponding functions of *Shape*. The function *Circle :: rotate()* does nothing since a circle does not change when being rotated, and *Circle :: is\_closed()* returns *true* since a circle is a closed curve. *Circle* is not an abstract class since it does not contain pure virtual functions, and consequently objects of *Circle* can be created.

A pure virtual function that is not defined in a derived class remains a pure virtual function. Such a derived class is also an abstract class. For example,

```

class Polygon: public Shape {    // abstract class
public:
    bool is_closed() { return true; }
};

```

```

        // override Shape::is_closed()
        // but draw() & rotate() still remain undefined.
};

class Triangle: public Polygon { // not abstract any more
private:
    Pt2d* vertices;
public:
    Triangle(Pt2d*);
    ~Triangle() { delete[] vertices; }
    void rotate(int);           // override Shape::rotate()
    void draw();                // override Shape::draw()
};

```

Class *Polygon* remains an abstract class since it has pure virtual functions *draw()* and *rotate()*, which are inherited from its base class *Shape* but have not been defined. Thus objects of *Polygon* can not be created. However, *Triangle* is no longer an abstract class since it does not contain any pure virtual functions (definitions of its members *rotate()* and *draw()* are omitted here) and objects of *Triangle* can be created.

Note that a virtual destructor is defined for the base class *Shape*. This will ensure proper cleanup for the derived class *Triangle*, which frees dynamic memory space in its destructor. In general, a class with a virtual function should have a virtual destructor, although other base classes may also need one; see the example in §8.1.6.

Below is a design problem for iterative numeric methods on solving linear systems of algebraic equations  $Ax = b$ , where  $A$  is a square matrix,  $b$  is the right-hand side vector, and  $x$  is the unknown vector. In §6.6 the conjugate gradient (CG) method is implemented for a Hermitian and positive definite matrix  $A$  and in §11.3 the generalized minimum residual (GMRES) method is introduced for any nonsingular matrix  $A$ . These two iterative methods require only matrix-vector multiplication and some vector operations in order to solve the linear system  $Ax = b$ . Details of CG and GMRES are not needed here. Instead, a class hierarchy is designed to apply CG and GMRES to full, band, and sparse matrices  $A$ . In some applications, most entries of the matrix  $A$  are not zero and all entries of  $A$  are stored. Such a matrix storage format is called a *full matrix*. In some other applications, the entries of  $A$  are zero outside a band along the main diagonal of  $A$ . Only entries within the band (zero or nonzero) may be stored to save memory and such a format is called a *band matrix*. Yet in other applications most entries of  $A$  are zero and only nonzero entries are stored to further save memory; such a format is called a *sparse matrix*. Three different classes need be defined for these three matrix storage formats, of which the full matrix format is given in §6.3 where every entry of a matrix is stored.

Details of these matrix storage formats are not needed here, although they can be found in §11.1.

The objective of such a design is that the CG and GMRES methods are defined only once, but are good for all three matrix storage formats, instead of providing one definition for each storage format. This should also be extendible possibly by other users later to other matrix formats such as symmetric sparse, band, and full matrices (only half of the entries need be stored for symmetric matrices to save memory). Since CG and GMRES depend on a matrix-vector product, which must be done differently for each matrix storage format, the correct matrix-vector multiplication function has to be called for a particular matrix format. To provide only one definition for CG and GMRES, they can be defined for a base class and inherited for derived classes representing different matrix formats. To ensure the correct binding of matrix-vector multiplication to each matrix storage format, a virtual function can be utilized. To be more specific, define a base class

```
class AbsMatrix {                                // base class
public:
    virtual ~AbsMatrix() { }                    // a virtual destructor

    virtual Vtr operator*(const Vtr &) const = 0;
                                                // matrix vector multiply

    int CG();
    int GMRES();
};

int AbsMatrix::CG() { /* ... definition omitted here */ }
int AbsMatrix::GMRES() { /* ... definition omitted here */ }
```

where *Vtr* is a vector class as defined in Chapter 6, which can also be thought of as the standard *vector<double>* here. This matrix class serves as an interface for all derived classes. Since there is not enough information to define the matrix-vector multiplication operator *\**, it is declared as a pure virtual function, which will be overridden in the derived classes. For such a class, it is natural to define a virtual destructor to ensure that objects of derived classes are properly cleaned up, although this virtual destructor does nothing. The class *AbsMatrix* is actually an abstract class so that no objects of *AbsMatrix* can be created. With the matrix-vector multiply operator *\** (although it is not defined yet), the functions *CG()* and *GMRES()* can be fully defined, which can be inherited and used in derived classes once the matrix-vector multiply operator *\** is defined. For simplicity, the arguments of *CG()* and *GMRES()* and their definitions are not given here (nor are they needed since only the design problem is considered in this section).

Derived classes for full, band, and sparse matrices can now be defined:

```

class FullMatrix: public AbsMatrix {
public:      // ... in addition to other members
    Vtr operator*(const Vtr&) const;
            // multiply a full matrix with a vector
};

class SparseMatrix: public AbsMatrix {
public:      // ... in addition to other members
    Vtr operator*(const Vtr&) const;
            // multiply a sparse matrix with a vector
};

class BandMatrix: public AbsMatrix {
public:      // ... in addition to other members
    Vtr operator*(const Vtr&) const;
            // multiply a band matrix with a vector
};

```

The matrix-vector multiply operator `*` can be defined (but omitted here) for each of the derived classes and thus `CG()` and `GMRES()` can be invoked for objects of *FullMatrix*, *BandMatrix*, and *SparseMatrix*. For example,

```

void f(FullMatrix& fm, BandMatrix& bm, SparseMatrix& sm) {
    fm.CG();           // call CG() on FullMatrix fm
    bm.CG();           // call CG() on BandMatrix bm
    sm.CG();           // call CG() on SparseMatrix sm
}

void g(vector<AbsMatrix*>& vm) {
    for (int i = 0; i < vm.size(); i++) {
        vm[i]->GMRES();
        // the object pointed to by vm[i] could be FullMatrix,
        // BandMatrix, or SparseMatrix. Its exact type may not
        // be known at compile-time.
    }
}

```

During such an invocation, the matrix-vector multiply operator `*` inside `CG()` and `GMRES()` is guaranteed to be called correctly according to the type of the objects, since the operator `*` is declared to be a *virtual* function in the base class. Note that in the call `vm[i]->GMRES()`, `vm[i]` may be a pointer to *FullMatrix*, *BandMatrix*, or *SparseMatrix*. This technique can be combined with templates so that one definition of `CG()` and `GMRES()` can be applied to different matrix storage formats with different precisions for real and complex matrices. This is the main subject of Chapter 11.

## 8.3 Access Control

### 8.3.1 Access to Members

With derived classes, the keyword *protected* is introduced. A member (it can be a function, type, constant, etc., as well as a data member) of a class can be declared to be *private*, *protected*, or *public*.

- A *private* member can be used only by members and friends of the class in which it is declared,
- A *protected* member can be used by members and friends of the class in which it is declared. Furthermore, it becomes a private, protected, or public member of classes derived from this class; as a member of a derived class, it can be used by members and friends of the derived class and maybe by other members in the class hierarchy depending on the form of inheritance. See §8.3.2 for more details.
- A *public* member can be used freely in the program. It also becomes a private, protected, or public member of classes derived from this class, depending on the form of inheritance. See §8.3.2 for more details.

For example,

```
class B {
private:
    int i;                // a private member
protected:
    float f;              // a protected member
public:
    double d;             // a public member
    void g1(B& b) { f = b.f; } // f and b.f are accessible
};

void g() {
    B bb;                 // by default constructor
    bb.i = 5;             // error, can not access bb.i
    bb.f = 3.14;          // error, can not access bb.f
    bb.d = 2.71;          // bb.d is accessible freely
}
```

The specifiers *protected* and *private* mean the same when inheritance is not involved.

Protected members of a class are designed for use by derived classes and are not intended for general use. They provide another layer of information hiding, similar to private members. The protected part of a class usually provides operations for use in the derived classes, and normally does not contain data members since it can be more easily accessed or abused than



the private part. For this reason the data member  $x$  in the class  $Pt$  in §8.1 is better kept *private* than *protected*. Despite this, the derived classes  $Pt2d$  and  $Pt3d$  can still print out the value of  $x$ .

### 8.3.2 Access to Base Classes

A base class can be declared *private*, *protected*, or *public*, in defining a derived class. For example,

```
class B { /* ... */ };           // a class
class X: public B { /* ... */ }; // B is a public base
class Y: protected B { /* ... */ }; // B is a protected base
class Z: private B { /* ... */ }; // B is a private base
```

**Public** derivation makes the derived class a subtype of its base class and is the most common form of derivation. **Protected** and **private** derivations are used to represent implementation details. No matter what form the derivation is, a derived class contains all the members of a base class (some of which may have been overridden), although members inherited from the private part of the base class are not directly accessible even by members and friends of the derived class (they are called **invisible members** of the derived class in §8.1.1). The form of derivation controls access to the derived class's members inherited from the base class and controls conversion of pointers and references from the derived class to the base class. They are defined as follows. Suppose that class  $D$  is derived from class  $B$ .

- If  $B$  is a private base, its public and protected members become private members of  $D$ . Only members and friends of  $D$  can convert a  $D^*$  to a  $B^*$ .
- If  $B$  is a protected base, its public and protected members become protected members of  $D$ . Only members and friends of  $D$  and members and friends of classes derived from  $D$  can convert a  $D^*$  to a  $B^*$ .
- If  $B$  is a public base, its public members become public members of  $D$  and its protected members become protected members of  $D$ . Any function can convert a  $D^*$  to a  $B^*$ . In this case,  $D$  is called a *subtype* of  $B$ .

The following example illustrates the use of protected members and public derivation.

```
class B {
private:
    int i;
protected:
    float f;
```

```

public:
    double d;
    void g1(B b) { f = b.f; }    // f and b.f are accessible
};                               // by a member of B

class X: public B {             // public derivation
protected:
    short s;
public:
    void g2(X b) { f = b.f; }    // f and b.f are accessible
};                               // by members of X

```

Since  $X$  has a public base  $B$ , it has **three public members**:  $d$  and  $g1()$  (inherited from  $B$ ), and  $g2()$  (its own), **two protected members**:  $f$  (inherited from  $B$ ) and  $s$  (its own), but **no** private members.  $X$  also inherits a member  $i$  from base  $B$ , but this member is invisible in  $X$ . The protected member  $f$  of class  $B$  and member  $f$  of the object  $b$  of type  $B$  are accessible in the member function  $B::g1(B\ b)$ . Similarly, the protected (inherited from base class) member  $f$  of class  $X$  and member  $f$  of the object  $b$  of type  $X$  are accessible in the member function  $X::g2(X\ b)$ .

However, it is illegal to define a member function  $g3()$  of  $X$  this way:

```

class X: public B {             // public derivation
protected:
    short s;
public:
    void g2(X b) { f = b.f; }    // f and b.f are accessible
    void g3(B b) { f = b.f; }    // error: b.f not accessible
};

```

In the definition of  $X::g3(B\ b)$ , the protected member  $f$  of object  $b$  of type  $B$  is not accessible. Member  $b.f$  of object  $b$  of type  $B$  is only accessible by members and friends of class  $B$ . Note the only difference between  $X::g2(X)$  and  $X::g3(B)$  is the type of their arguments.

A derived class inherits members (e.g.,  $B::f$  above) that are *protected* and *public* in its base class. These inherited members (e.g.,  $X::f$ ) of the derived class are accessible by members and friends of the derived class. However, **the derived class can not access protected members** (e.g.,  $b.f$  in the definition of  $X::g3(B)$ ) of an object of the base class, just as it can not access protected members of objects of any other class.

No matter what type of inheritance it is, a private member of a base class can not be used in derived classes to achieve a strong sense of **information hiding**, since otherwise a private member of any class could be accessed and possibly abused easily by defining a derived class from it.

Access to an inherited member may be adjusted to that of the base class by declaring it *public* or *protected* in the derived class using the scope resolution operator `::` as in the following example.

```
class B {
private:
    int i;
protected:
    float f;
public:
    double d;
};

class Y: protected B {    // protected derivation
protected:
    short s;
public:
    B::d;                  // B::d is adjusted to be public
};
```

Now the member *d* of *Y*, inherited from the protected base *B*, is public in *Y*, instead of *protected* by default. Note that the private member *i* of *B* can not be adjusted in *Y*, since it is not visible in *Y*.

If a derived class adds a member with the same name as a member of a base class, the new member in the derived class hides the inherited member from the base class. The inherited member may be accessible using the scope resolution operator preceded by the base class name. For example,

```
class B {
public:
    double d;
};

class Z: private B {        // private derivation
private:
    double d;              // this d hides B::d
public:
    void f(Z& z) {
        z.d = 3.1415926;    // Z::d instead of B::d
        z.B::d = 2.718;    // z's member B::d
    }
};
```

The derivation specifier for a base class can be left out. In this case, the base defaults to a private base for a class and a public base for a struct:

```
class XX: B { /* ... */ };    // B is a private base
```

A class can be directly derived from two or more base classes. This is called *multiple inheritance*. In contrast, derivation from only one direct base class is called *single inheritance*. This section talks about multiple inheritance. Consider a class *Circle\_in\_Triangle* (circle inscribed in a triangle) derived from classes *Circle* and *Triangle* :

The derived class *Circle\_in\_Triangle* inherits properties from both *Circle* and *Triangle*. It can be used as

```
void f(Circle_in_Triangle& ct) {
    ct.dilate(5.0);           // call Circle::dilate()
    ct.refine();               // call Triangle::refine()
    ct.draw();                 // call Circle_in_Triangle::draw()
```

```

}

double curvature(Circle*);           // curvature of a curve
vector<double> angles(Triangle*);    // angles of a triangle

void g(Circle_in_Triangle* pct) {
    double c = curvature(pct);
    vector<double> a = angles(pct);
}

```

Since public derivation is used for both base classes, any function can convert a *Circle\_in\_Triangle\** to *Circle\** or *Triangle\**.

#### 8.4.1 Ambiguity Resolution

When two base classes have members with the same name, they can be resolved by using the scope resolution operator. For example, both *Circle* and *Triangle* have a function named *area()*. They must be referred to with the class names:

```

void h(Circle_in_Triangle* pct) {
    double ac = pct->Circle::area();    // OK
    double at = pct->Triangle::area();  // OK
    double aa = pct->area();             // ambiguous, error
    pct->draw();                         // OK
}

```

Overload resolution is not applied across different types. In particular, ambiguities between functions from different base classes are not resolved based on argument types. A *using*-declaration can bring different functions from base classes to a derived class and then overload resolution can be applied. For example,

```

class A {
public:
    int g(int);
    float g(float);
};

class B {
public:
    char g(char);
    long g(long);
};

class C: public A, public B {
public:

```

```

    using A::g;                // bring g() from A
    using B::g;                // bring g() from B
    char g(char);              // it hides B::g(char)
    C g(C);
};

void h(C& c) {
    c.g(c);                    // C::g(C) is called
    c.g(1);                    // A::g(int) is called
    c.g(1L);                   // B::g(long) is called
    c.g('E');                  // C::g(char) is called
    c.g(2.0);                  // A::g(float) is called
}

```

#### 8.4.2 Replicated Base Classes

With the possibility of derivation from two bases, a class can be a base twice. For example, if both *Circle* and *Triangle* are derived from a class *Shape*, then the base class *Shape* will be inherited twice in the class *Circle\_in\_Triangle*:

```

class Shape {
protected:
    int color;                  // color of shape
public:
    // ...
    virtual void draw() = 0;    // a pure virtual function
};

class Circle: public Shape {
    // ...                     // inherit color for Circle
public:
    void draw();               // draw circle
};

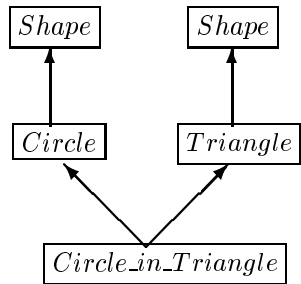
class Triangle: public Shape {
    // ...                     // color for Triangle
public:
    void draw();               // draw the triangle
};

class Circle_in_Triangle: public Circle, public Triangle {
    // ...                     // two bases
public:
    void draw();               // override Circle::draw()
}

```

```
};                                     // and Triangle::draw()
```

This causes no problems if the colors of a *Circle* and a *Triangle* for an object of *Circle\_in\_Triangle* can be different. Indeed, two copies of *Shape* are needed to store the colors. This class hierarchy can be represented as



To refer to members of a replicated base class, the scope resolution operator must be used. For example,

```
void Circle_in_Triangle::draw() {
    int cc = Circle::color;           // or Circle::Shape::color
    int ct = Triangle::color;         // or Triangle::Shape::color
    // ...
    Circle::draw();
    Triangle::draw();
}
```

A virtual function of a replicated base class can be overridden by a single function in a derived class. For example, *Circle\_in\_Triangle::draw()* overrides *Shape::draw()* from the two copies of *Shape*.

### 8.4.3 Virtual Base Classes

Often a base class need not be replicated. That is, only one copy of a replicated class need be inherited for a derived class object. This can be done by specifying the base to be *virtual*. Every *virtual* base of a derived class is represented by the same (shared) object. For example, if the circle and triangle in *Circle\_in\_Triangle* must have the same color, then only one copy of *Shape* is needed for storing the color information. The base class *Shape* then should be declared to be *virtual* :

```
class Shape {
    int color;                       // color of shape
public:
    // ...
    virtual void draw() = 0;         // pure virtual function
};
```

```

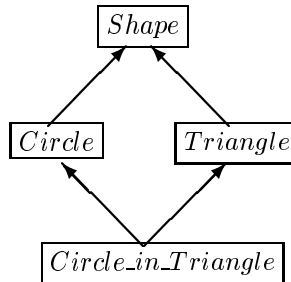
class Circle: public virtual Shape { // virtual base Shape
    // ...                          // inherit color
public:                             // for Circle
    void draw();                    // draw circle
};

class Triangle: public virtual Shape { // virtual base Shape
    // ...                          // inherit color
public:                             // for Triangle
    void draw();                    // draw triangle
};

class Circle_in_Triangle: public Circle, public Triangle {
    // ...                          // two bases
public:
    void draw();                    // override Circle::draw()
                                    // and Triangle::draw()
};

```

This class hierarchy can be represented as



Compare this diagram with the inheritance graph in §8.4.2 to see the difference between **ordinary inheritance** and **virtual inheritance**. In an inheritance graph, every base class that is specified to be *virtual* will be represented by a single object of that class. The language ensures that a constructor of a virtual base class is called exactly once.

#### 8.4.4 Access Control in Multiple Inheritance

A name or a base class is called accessible if it can be reached through any path in a multiple inheritance graph, although often it can be reached through several paths. For example,

```

class B {
public:
    double d;

```



```

    static double sdm;
};

class D1: public virtual B { /* ... */ };
class D2: public virtual B { /* ... */ };
class D12: protected D1, public D2 { /* ... */ };

```

```

D12* pd12 = new D12;
B* pb = pd12;           // accessible through public base D2
double m = pd12->d;      // accessible through public base D2

```

The member *pd12*→*d* is accessible publicly since *D12* has a public base *D2* and *d* is a public member of *D2*. It would be inaccessible if both *D1* and *D2* were protected bases. However, ambiguities might arise when a single entity was accessible through more than one path. Again, the scope resolution operator can be used to resolve such ambiguities. For example,

```

class X1: public B { /* ... */ };
class X2: public B { /* ... */ };
class X12: protected X1, private X2 {
    void f();
};

void X12::f() {
    X12* p = new X12;
    // ... assign some value to *p
    double i = p->d;           // illegal, ambiguous
                                // X12::X1::B::d or X12::X2::B::d?

    double j = p->X1::B::d;    // OK
    double k = p->X2::B::d;    // OK

    double n = p->sdm;         // OK
                                // only one B::sdm in an X12 object
}

```

There are two copies of member *d* in an object of *X12*, but there is only one static member *sdm* in *X12*.

## 8.5 Run-Time Type Information

Due to run-time polymorphism and assignment of pointers or references from one type to another, the type of an object may be lost. Recovering the run-time type information (or RTTI for short) of an object requires the

system to examine the object to reveal its type. This section introduces two mechanisms that can be used to recover the run-time type of an object.

### 8.5.1 The *dynamic\_cast* Mechanism

The *dynamic\_cast* mechanism can be used to handle cases in which the correctness of a conversion can not be determined by the compiler. Suppose *p* is a pointer. Then

```
dynamic_cast<T*>(p)
```

examines the object pointed to by *p* at run-time. If this object is of class *T* or has a unique base class of type *T*, then it returns a pointer of type *T\** to the object; otherwise it returns 0. If the value of *p* is 0 (null pointer), then *dynamic\_cast<T\*>(p)* returns 0. For example,

```
class A { /* ... */ };
class B { /* ... */ };
class C: public A, protected B { /* ... */ };

void f(C* p) {
    A* q1 = p;                // OK
    A* q2 = dynamic_cast<A*>(p); // OK

    B* p1 = p;                // error, B is protected base
    B* p2 = dynamic_cast<B*>(p); // OK, but 0 is assigned to p2
}

void g(B* pb) {               // assume A, B have virtual functions
    if (A* pa = dynamic_cast<A*>(pb)) {
        // if pb points to an object of type A, do something
    } else {
        // if pb does not point to an A object, do something else
    }
}
```

Since *B* is a protected base of *C*, function *f()* can not directly convert a pointer to *C* into a pointer to *B*. Although using *dynamic\_cast* makes it legal, the null pointer is returned. In *g()*, converting *B\** into *A\** requires that *A* and *B* have virtual functions so that a variable of *A\** or *B\** may be of another type, for example, *C\**.

Casting from a derived class to a base class is called an *upcast* because of the convention of drawing an inheritance graph growing from the base class downwards. Similarly, a cast from a base class to a derived class is called a *downcast*. A cast from a base to a sibling class (like the cast from *B* to *A* above) is called a *crosscast*. A *dynamic\_cast* requires a pointer or reference to a polymorphic type to do a downcast or crosscast. The result of

the return value of a *dynamic\_cast* of a pointer should always be **explicitly tested**, as in function *g()* above. If the result is 0, it means a failure in the conversion indicated.

For a reference *r*, then

```
dynamic_cast<T&>(r);
```

tests to see if the object referred to by *r* is of type *T*. When it is not, it throws a **bad\_cast** exception. To test a successful or failed cast using references, a suitable handler should be provided; see Chapter 9 for details. For example,

```
void h(B* pb, B& rb) { // assume A, B have virtual functions
    A* pa = dynamic_cast<A*>(pb);
    if (pa) {           // when $pa$ is not null pointer
        // if pb points to an A object, do something
    } else {            // when $pa$ is null pointer
        // if pb does not point to an A object, do something else
    }

    A& ra = dynamic_cast<A&>(rb); // rb refers to an A object?
}

int main() {
    try{
        A* pa = new A;
        B* pb = new B;
        h(pa, *pa);
        h(pb, *pb);
    } catch (bad_cast) { // exception handler
        // dynamic_cast<A&> failed, do something.
    }
}
```

A **dynamic\_cast** can cast from a polymorphic virtual base class to a derived class or a sibling class, but a **static\_cast** can not since it does not examine the object from which it casts. However, *dynamic\_cast* can not cast from a *void\** (but *static\_cast* can) since nothing can be assumed about the memory pointed to by a *void\**. Both *dynamic\_cast* and *static\_cast* can not cast away *const*, which requires a **const\_cast**. For example,

```
void cast(const A* p, A* q) {
    q = const_cast<A*>(p); // OK
    q = static_cast<A*>(p); // error, can not cast away const
    q = dynamic_cast<A*>(p); // error, can not cast away const
}
```

### 8.5.2 The typeid Mechanism

The *dynamic\_cast* operator is enough for most problems that need to know the type of an object at run-time. However, situations occasionally arise when the exact type of an object needs to be known. The *typeid* operator yields an object representing the type of its operand; it returns a reference to a standard library class called *type\_info* defined in header `<typeinfo>`. The operator *typeid*() can take a type name or an expression as its operand and returns a reference to a *type\_info* object representing the type name or the type of the object denoted by the expression. If the value of a pointer or a reference operand is 0, then *typeid*() throws a *bad\_typeid* exception. For example,

```
void f(Circle* p, Triangle& r, Circle_in_Triangle& ct) {
    typeid(r);           // type of object referred to by r
    typeid(*p);          // type of object pointed to by p

    if (typeid(ct) == typeid(Triangle)) { /* ... */ }
    if (typeid(r) != typeid(*p)) { /* ... */ }

    cout << typeid(r).name();           // print out type name
}
```

The function *type\_info::name*() returns a character string representing the type name. Equality and inequality of objects of *type\_info* can be compared. The implementation-independent part of the class *type\_info*, presented in the standard header `<typeinfo>`, has the form:

```
class type_info {
private:    // prevent copy initialization and assignment
    type_info(const type_info&);
    type_info& operator=(const type_info&);
public:
    virtual ~typeid();           //polymorphic type
    bool operator==(const type_info&) const; // can be compared
    bool operator!=(const type_info&) const; // can be compared
    bool before(const type_info&) const;     // can be sorted
    const char* name() const;               // name of type
};
```

There can be more than one *type\_info* object for each type in a system. Thus comparisons for equality and inequality should be applied to *type\_info* objects instead of their pointers or references.

### 8.5.3 Run-Time Overhead

Using run-time type information (*dynamic\_cast* or *typeid*) involves overhead since the system examines the type of an object. It should and can be avoided in many cases in which virtual functions suffice. Use virtual functions rather than *dynamic\_cast* or *typeid* if possible since **virtual functions normally introduce less run-time overhead**. Even a virtual function can cause a lot of performance penalty when it is small (i.e., it does not perform a lot of operations) and called frequently (e.g., inside a large loop). **When a virtual function contains a lot of code** (i.e., performs a lot of computation) or is not called frequently, the overhead of virtual function dispatch will be insignificant compared to the overall computation.

In large-scale computations, **static (compile-time) checking may be preferred** where applicable, since it is safer and imposes less overhead than run-time polymorphism. On one hand, run-time polymorphism is the cornerstone of object-oriented programming. On the other hand, too much run-time type checking can also be a disadvantage due to the run-time overhead it imposes, especially in performance-sensitive computations. In §8.6, some techniques are described to replace certain virtual functions by using templates.

## 8.6 Replacing Virtual Functions by Static Polymorphism

Virtual functions provide a nice design technique in software engineering. However, virtual functions may slow down a program when they are called frequently, especially when they do not contain a lot of instructions. In this section, two techniques are presented to replace certain virtual functions by static polymorphism, in order to improve run-time efficiency.

The idea of the first technique [BN94] is to define a base template class and define functions operating on this base class. When defining a derived class from the base template class, the template parameter of this base class is taken to be the type of the derived class. Then the functions defined for the base class can be used for derived classes as well.

Consider the example in which a function *CG()* is defined for a base matrix class but may be called on derived classes for full, band, and sparse matrices. A virtual function is used in §8.2 to define matrix-vector multiplication, upon which the function *CG()* depends. Then this virtual function is overridden in derived classes for full, band, and sparse matrices, which enables one version of *CG()* to be called for different derived classes. Here the same goal can be achieved without using a virtual function. Rather, a matrix-vector multiplication is defined for a base template class that refers to the template parameter, which will be a derived class. The following code illustrates this idea.

```

class Vtr { /* a vector class */ };

template<class T> class Matrix { // base class
    const T& ReferToDerived() const {
        return static_cast<const T&>(*this);
    }
public:

    // refer operation to derived class
    Vtr operator*(const Vtr& v) const {
        return ReferToDerived()*v;
    }

    // define common functionality in base class that can
    // be called on derived classes
    void CG(const Vtr& v) { // define CG here
        Vtr w = (*this)*v; // call matrix vector multiply
        cout << "calling CG\n";
    }
};

class FullMatrix: public Matrix<FullMatrix> {
// encapsulate storage information for full matrix
public:
    Vtr operator*(const Vtr& v) const {
        // define (full) matrix vector multiply here
        cout << "calling full matrix vector multiply\n";
    }
};

class BandMatrix: public Matrix<BandMatrix> {
// encapsulate storage information for band matrix
public:
    Vtr operator*(const Vtr& v) const {
        // define (band) matrix vector multiply here
        cout << "calling band matrix vector multiply\n";
    }
};

class SparseMatrix: public Matrix<SparseMatrix> {
// encapsulate storage information for sparse matrix
public:
    Vtr operator*(const Vtr& v) const {
        // define (sparse) matrix vector multiply here
        cout << "calling sparse matrix vector multiply\n";
    }
};

```

```

    }
};

void f(const Vtr& v) {
    FullMatrix A;
    A.CG(v);           // Calling CG() on full matrix

    BandMatrix B;
    B.CG(v);           // Calling CG() on band matrix

    SparseMatrix S;
    S.CG(v);           // Calling CG() on sparse matrix
}

```

Although the code above compiles and runs, it has been greatly simplified and full definitions of *CG()* and matrix-vector multiply functions are not given. The function *CG()* is defined for the base class *Matrix*, which depends on the matrix-vector multiply operator *\**. The operator *\** is syntactically defined for the base *Matrix*, but actually refers to the operator *\** belonging to the template parameter *T*. When *T* is instantiated by, for example, *FullMatrix* in the definition of the class *FullMatrix*, this operator *\** now is the *FullMatrix*-vector multiply operator, which can be fully defined according to its matrix structure.

In this approach, the type of an object is known at compile-time and thus there is no need for virtual function dispatch. One version of the function *CG()* can be called for different classes, which is achieved without using virtual functions.

A simple but more concrete and complete example using this technique is now given. First define a function *sum()* for a base template matrix that adds all entries of a matrix. Then define a derived class for full matrices that stores all entries of a matrix and another derived class for symmetric matrices that stores only the lower triangular part of a matrix to save memory. The goal is to give only one version of *sum()* that can be called for full and symmetric matrices, without using virtual functions. This can be achieved as

```

template<class T> class Mtx {    // base matrix
private:
    // refer to derived class
    T& ReferToDerived() {
        return static_cast<T&>(*this);
    }

    // entry() uses features of derived class
    double& entry(int i, int j) {
        return ReferToDerived()(i,j);
    }
};

```

```

    }
protected:
    int dimn;          // dimension of matrix
public:
    // define common functionality in base class that can
    // be called on derived classes
    double sum() {      // sum all entries
        double d = 0;
        for (int i = 0; i < dimn; i++)
            for (int j = 0; j < dimn; j++) d += entry(i,j);
        return d;
    }
};

class FullMtx: public Mtx<FullMtx> {
    double** mx;
public:
    FullMtx(int n) {
        dimn = n;
        mx = new double* [dimn];
        for (int i=0; i<dimn; i++) mx[i] = new double [dimn];
        for (int i=0; i<dimn; i++)
            for (int j=0; j<dimn; j++)
                mx[i][j] = 0;          // initialization
    }
    double& operator()(int i, int j) { return mx[i][j]; }
};

class SymmetricMtx: public Mtx<SymmetricMtx> {
    // store only lower triangular part to save memory
    double** mx;
public:
    SymmetricMtx(int n) {
        dimn = n;
        mx = new double* [dimn];
        for (int i=0; i<dimn; i++) mx[i] = new double [i+1];
        for (int i=0; i<dimn; i++)
            for (int j = 0; j <= i; j++)
                mx[i][j] = 0;          // initialization
    }
    double& operator()(int i, int j) {
        if (i >= j ) return mx[i][j];
        else return mx[j][i];          // due to symmetry
    }
};

```



```

void g() {
    FullMtx A(2);
    A(0,0) = 5; A(0,1) = 3; A(1,0) = 3; A(1,1) = 6;
    cout << "sum of full matrix A = " << A.sum() << '\n';

    SymmetricMtx S(2);    // just assign lower triangular part
    S(0,0) = 5; S(1,0) = 3; S(1,1) = 6;
    cout << "sum of symmetric matrix S = " << S.sum() << '\n';
}

```

Here the member function *sum()* depends on *entry(i, j)* that refers to an entry at row *i* and column *j* of a matrix through a function call operator (). But *entry()* is not defined for *FullMtx* and *SymmetricMtx*, and the function call operator () is not defined for the base class *Mtx*.

The second technique in this section does not use inheritance at all, but only uses templates. It is much simpler in many situations. Taking the matrix solver *CG()* as an example, this technique defines *CG()* as a global template function whose template parameter can be any type that defines a multiply operator with a vector:

```

class Vtr {    /* a vector class */ };

template<class M>    // define CG as a global function
void CG(const M& m, const Vtr& v) {
    // define CG here for matrix m and vector v
    Vtr w = m*v;    // call matrix vector multiply
    cout << "calling CG\n";
}

class FullMatrix {
//    encapsulate storage information for full matrix
public:
    Vtr operator*(const Vtr& v) const {
        // define (full) matrix vector multiply here
        cout << "calling full matrix vector multiply\n";
    }
};

class BandMatrix {
//    encapsulate storage information for band matrix
public:
    Vtr operator*(const Vtr& v) const {
        // define (band) matrix vector multiply here
        cout << "calling band matrix vector multiply\n";
    }
}

```

```

};

class SparseMatrix {
// encapsulate storage information for sparse matrix
public:
    Vtr operator*(const Vtr& v) const {
        // define (sparse) matrix vector multiply here
        cout << "calling sparse matrix vector multiply\n";
    }
};

void ff(const Vtr& v) {
    FullMatrix A;
    CG(A, v);           // Calling CG() on full matrix

    BandMatrix B;
    CG(B, v);           // Calling CG() on band matrix

    SparseMatrix S;
    CG(S, v);           // Calling CG() on sparse matrix
}

```

If this technique is applied to the second example above, where the entries of a matrix are *sum()*ed, each of the classes for *FullMtx* and *SymmetricMtx* will need to define a function *entry()*:

```

template<class M> double sum(const M& m) {
    double d = 0;
    for (int i = 0; i < m.dimn; i++)
        for (int j = 0; j < m.dimn; j++) d += m.entry(i,j);
    return d;
}

class FullMtx {
    double** mx;
public:
    FullMtx(int n) {
        dimn = n;
        mx = new double* [dimn];
        for (int i=0; i<dimn; i++) mx[i] = new double [dimn];
        for (int i=0; i<dimn; i++)
            for (int j=0; j<dimn; j++)
                mx[i][j] = 0;           // initialization
    }
    int dimn;
    double& operator()(int i, int j) { return mx[i][j]; }
}

```

```

double entry(int i, int j) const {
    return const_cast<FullMtx&>(*this)(i,j);
}
};

class SymmetricMtx {
    // store only lower triangular part to save memory
    double** mx;
public:
    SymmetricMtx(int n) {
        dimn = n;
        mx = new double* [dimn];
        for (int i=0; i<dimn; i++) mx[i] = new double [i+1];
        for (int i=0; i<dimn; i++)
            for (int j = 0; j <= i; j++)
                mx[i][j] = 0;          // initialization
    }
    int dimn;
    double& operator()(int i, int j) {
        if (i >= j ) return mx[i][j];
        else return mx[j][i];        // due to symmetry
    }
    double entry(int i, int j) const {
        return const_cast<SymmetricMtx&>(*this)(i,j);
    }
};

void gg() {
    FullMtx A(2);
    A(0,0) = 5; A(0,1) = 3; A(1,0) = 3; A(1,1) = 6;
    cout << "sum of full matrix A = " << sum(A) << '\n';

    SymmetricMtx S(2);    // just assign lower triangular part
    S(0,0) = 5; S(1,0) = 3; S(1,1) = 6;
    cout << "sum of symmetric matrix S = " << sum(S) << '\n';
}

```

When function *entry()* is large and accesses local information of *FullMtx* and *SymmetricMtx*, this may cause a lot of code duplication. In this case, the first technique might be better. This example is only for illustrating the idea; if just for summing entries of a matrix, the function *entry()* may not be needed.

Using virtual functions as shown in §8.2, the derived classes *FullMatrix*, *BandMatrix*, and *SparseMatrix* may not have to be known when the base class *AbsMatrix* is compiled. That is, if one wants to add one more

class *SymmetricMatrix* to this class hierarchy, the code for the base class *AbsMatrix* need not be recompiled. However, using the techniques here without virtual functions, the complete type of all classes must be known at compile-time. Thus, in this aspect, there may be some design disadvantage to static polymorphism. Besides, not all virtual functions can be replaced by templates.

## 8.7 Exercises

- 8.7.1. Implement the class hierarchy for points in one, two, and three dimensions, as outlined in §8.1. Add more functions such as moving a point, finding the equation of the line passing through two points, and evaluating the distance between two points. Create a vector of (pointers to) points and print them out using run-time polymorphism.
- 8.7.2. Implement the class hierarchy for points in one, two, and three dimensions, as outlined in §8.1, using pointers for their data members. For example, declare a pointer to double for the  $x$ -coordinate in *Pt* and for the  $y$ -coordinate in *Pt2d*. Consequently, user-defined copy constructor and copy assignment need be defined for them, and a virtual destructor is needed for the base class *Pt*.
- 8.7.3. When should a class have a virtual destructor?
- 8.7.4. Rewrite the base class *B* and derived class *D* in §8.1.6 so that *B* will allocate space for  $d$  doubles in its constructor and *D* for  $i$  integers, where  $d$  and  $i$  must be passed as arguments to the constructors.
- 8.7.5. Summarize all C++ language features that support polymorphism (in compile- and run-times), and discuss their advantages and disadvantages.
- 8.7.6. Define an abstract class called *Shape* containing a pure virtual function *distance()* that takes a two-dimensional point as argument and is meant to find the distance between the point and *Shape*. Derive three classes for *Triangle*, *Circle*, and *Rectangle*, respectively, and override the virtual function *distance()* in each of the derived classes. The distance between a point and a figure is defined to be the minimum distance between the given point and any point in the figure. Create a vector of pointers to *Shape* that actually point to objects of *Triangle*, *Circle*, or *Rectangle*, and create a vector of points. Call *distance()* to find the distance between each point to each of the *Shape* objects.
- 8.7.7. Test the class hierarchy for full, band, and sparse matrices as outlined in §8.2 in which one definition of *CG()* and *GMRES()* is given for

three matrix storage formats. Instead of writing code implementing each function, you may just let, for example,  $CG()$  call a matrix-vector multiply operator  $*$  and print out “Solving linear system using CG,” band-matrix vector multiply operator  $*$  print out “multiplying band matrix with vector,” and sparse-matrix vector multiply operator  $*$  print out “multiplying sparse matrix with vector.” This is enough to know if the right functions are called. Then create a vector of pointers to *AbsMatrix* which actually point to objects of *FullMatrix*, *BandMatrix*, or *SparseMatrix*, and call  $CG()$  and  $GMRES()$  on each of the matrix objects. Check the printouts to see if dynamic binding is working correctly.

- 8.7.8. Define an abstract class called *Shape* containing a pure virtual function *inside()* that takes a point as argument and is meant to check if *Shape* contains the point in a two-dimensional Euclidean space. Derive three classes for *Triangle*, *Circle*, and *Rectangle*, respectively, and override the virtual function *inside()* in each of the derived classes. Create a vector of pointers to *Shape* that actually point to objects of *Triangle*, *Circle*, or *Rectangle*, and create a vector of points. Call function *inside()* to see which point is inside which figure.

Here is a hint on how to check if a point  $p(x, y)$  is contained inside a triangle with vertices  $p_0(x_0, y_0)$ ,  $p_1(x_1, y_1)$ , and  $p_2(x_2, y_2)$ . Let  $\mathbf{q}$  in boldface be a vector from the origin to point  $q$  for any point  $q$ , and  $\mathbf{p}_i\mathbf{p}_j$  be a vector from point  $p_i$  to point  $p_j$ . Then  $\mathbf{p}$  can be represented as  $\mathbf{p} = \mathbf{p}_0 + \alpha\mathbf{p}_0\mathbf{p}_1 + \beta\mathbf{p}_0\mathbf{p}_2$ . Applying vector cross-product with  $\mathbf{p}_0\mathbf{p}_1$  to its both sides yields  $\mathbf{p} \times \mathbf{p}_0\mathbf{p}_1 = (\mathbf{p}_0 + \alpha\mathbf{p}_0\mathbf{p}_1 + \beta\mathbf{p}_0\mathbf{p}_2) \times \mathbf{p}_0\mathbf{p}_1 = \mathbf{p}_0 \times \mathbf{p}_0\mathbf{p}_1 + \beta\mathbf{p}_0\mathbf{p}_2 \times \mathbf{p}_0\mathbf{p}_1$ . That is,  $(\mathbf{p} - \mathbf{p}_0) \times \mathbf{p}_0\mathbf{p}_1 = \beta\mathbf{p}_0\mathbf{p}_2 \times \mathbf{p}_0\mathbf{p}_1$ . Using the definition of vector cross-products gives

$$\beta = \frac{(x - x_0)(y_1 - y_0) - (x_1 - x_0)(y - y_0)}{(x_2 - x_0)(y_1 - y_0) - (x_1 - x_0)(y_2 - y_0)}.$$

Now check to see if points  $p$  and  $p_2$  are on the same side of  $\mathbf{p}_0\mathbf{p}_1$  by checking if  $\beta > 0$ . Similarly, check to see if  $p$  and  $p_1$  are on the same side of  $\mathbf{p}_0\mathbf{p}_2$ , and if  $p$  and  $p_0$  are on the same side of  $\mathbf{p}_1\mathbf{p}_2$ . These three conditions are true if and only if  $p$  is inside the triangle.

- 8.7.9. Generalize Exercise 8.7.8 to three-dimensional Euclidean spaces with three derived classes for *tetrahedron*, *ball*, and *cube*, respectively. The idea of checking to see if a point is inside a triangle given in Exercise 8.7.8 applies to tetrahedra. Such techniques are often used in computer-aided geometric design.
- 8.7.10. Define an abstract class *Shape*, derive three classes *Triangle*, *Circle*, and *Rectangle* from it, and define a function

```
bool intersect(Shape* s, Shape* t);
```

that determines if the two shapes  $s$  and  $t$  overlap. To achieve this, suitable virtual functions may need be declared for the base class *Shape* and overridden in derived classes *Triangle*, *Circle*, and *Rectangle*. Hint: this is a so-called *double dispatch* problem. An answer to this exercise can be found in [Van98].

- 8.7.11. Run the codes in §8.6 to see if they work as expected. Rewrite the codes into more general templates with a template parameter for the type of the entries of the matrix, so that they can handle real and complex matrices in single, double, and other precisions.