CHAPTER 1

Preliminaries

1.0 Introduction

This book is supposed to teach you methods of numerical computing that are practical, efficient, and (insofar as possible) elegant. We presume throughout this book that you, the reader, have particular tasks that you want to get done. We view our job as educating you on how to proceed. Occasionally we may try to reroute you briefly onto a particularly beautiful side road; but by and large, we will guide you along main highways that lead to practical destinations.

Throughout this book, you will find us fearlessly editorializing, telling you what you should and shouldn't do. This prescriptive tone results from a conscious decision on our part, and we hope that you will not find it irritating. We do not claim that our advice is infallible! Rather, we are reacting against a tendency, in the textbook literature of computation, to discuss every possible method that has ever been invented, without ever offering a practical judgment on relative merit. We do, therefore, offer you our practical judgments whenever we can. As you gain experience, you will form your own opinion of how reliable our advice is. Be assured that it is not perfect!

We presume that you are able to read computer programs in C++. The question, "Why C++?", is a complicated one. For now, suffice it to say that we wanted a language with a C-like syntax in the small (because that is most universally readable by our audience), which had a rich set of facilities for object-oriented programming (because that is an emphasis of this third edition), and which was highly backward-compatible with some old, but established and well-tested, tricks in numerical programming. That pretty much led us to C++, although Java (and the closely related C#) were close contenders.

Honesty compels us to point out that in the 20-year history of *Numerical Recipes*, we have never been correct in our predictions about the future of programming languages for scientific programming, *not once*! At various times we convinced ourselves that the wave of the scientific future would be ... Fortran ... Pascal ... C ... Fortran 90 (or 95 or 2000) ... Mathematica ... Matlab ... C++ or Java .... Indeed, several of these enjoy continuing success and have significant followings (not including Pascal!). None, however, currently command a majority, or even a large plurality, of scientific users.

With this edition, we are no longer trying to predict the future of programming languages. Rather, we want a serviceable way of communicating ideas about scientific programming. We hope that these ideas transcend the language, C++, in which we are expressing them.

When we include programs in the text, they look like this:

calendar.h

```
void flmoon(const Int n, const Int nph, Int &jd, Doub &frac) {
```

Our routines begin with an introductory comment summarizing their purpose and explaining their calling sequence. This routine calculates the phases of the moon. Given an integer `n` and a code `nph` for the phase desired (`nph` = 0 for new moon, 1 for first quarter, 2 for full, 3 for last quarter), the routine returns the Julian Day Number `jd`, and the fractional part of a day `frac` to be added to it, of the n*th* such phase since January, 1900. Greenwich Mean Time is assumed.

```
    const Doub RAD=3.14159265358979238/180.0;
    Int i;
    Doub am,as,c,t,t2,xtra;
    c=n+nph/4.0;                          This is how we comment an individual line.
    t=c/1236.85;
    t2=t*t;
    as=359.2242+29.105356*c;                    You aren't really intended to understand
    am=306.0253+385.816918*c+0.010730*t2;           this algorithm, but it does work!
    jd=2415020+28*n+7*nph;
    xtra=0.75933+1.53058868*c+((1.178e-4)-(1.55e-7)*t)*t2;
    if (nph == 0 || nph == 2)
        xtra += (0.1734-3.93e-4*t)*sin(RAD*as)-0.4068*sin(RAD*am);
    else if (nph == 1 || nph == 3)
        xtra += (0.1721-4.0e-4*t)*sin(RAD*as)-0.6280*sin(RAD*am);
    else throw("nph is unknown in flmoon");       This indicates an error condition.
    i=Int(xtra >= 0.0 ? floor(xtra) : ceil(xtra-1.0));
    jd += i;
    frac=xtra-i;
}
```

Note our convention of handling all errors and exceptional cases with a statement like `throw("some error message");`. Since C++ has no built-in exception class for type `char*`, executing this statement results in a fairly rude program abort. However we will explain in §1.5.1 how to get a more elegant result without having to modify the source code.

1.0.1 *What* Numerical Recipes *Is Not*

We want to use the platform of this introductory section to emphasize what *Numerical Recipes* is *not*:

1. *Numerical Recipes* is not a textbook on programming, or on best programming practices, or on C++, or on software engineering. We are not opposed to good programming. We try to communicate good programming practices whenever we can — but only incidentally to our main purpose, which is to teach how practical numerical methods actually work. The unity of style and subordination of function to standardization that is necessary in a good programming (or software engineering) textbook is just not what we have in mind for this book. Each section in this book has as its focus a particular computational method. Our goal is to explain and illustrate *that* method as clearly as possible. No single programming style is best for all such methods, and, accordingly, our style varies from section to section.

2. *Numerical Recipes* is not a program library. That may surprise you if you are one of the many scientists and engineers who use our source code regularly. What

makes our code *not* a program library is that it demands a greater intellectual commitment from the user than a program library ought to do. If you haven't read a routine's accompanying section and gone through the routine line by line to understand how it works, then you use it at great peril! We consider this a feature, not a bug, because our primary purpose is to teach methods, not provide packaged solutions. This book does not include formal exercises, in part because we consider each section's code to be the exercise: If you can understand each line of the code, then you have probably mastered the section.

There are some fine commercial program libraries [1,2] and integrated numerical environments [3-5] available. Comparable free resources are available, both program libraries [6,7] and integrated environments [8-10]. When you want a packaged solution, we recommend that you use one of these. *Numerical Recipes* is intended as a cookbook for cooks, not a restaurant menu for diners.

1.0.2 Frequently Asked Questions

This section is for people who want to jump right in.

1. *How do I use NR routines with my own program?*

The easiest way is to put a bunch of `#include`'s at the top of your program. Always start with `nr3.h`, since that defines some necessary utility classes and functions (see §1.4 for a lot more about this). For example, here's how you compute the mean and variance of the Julian Day numbers of the first 20 full moons after January 1900. (Now *there's* a useful pair of quantities!)

```
#include "nr3.h"
#include "calendar.h"
#include "moment.h"

Int main(void) {
    const Int NTOT=20;
    Int i,jd,nph=2;
    Doub frac,ave,vrnce;
    VecDoub data(NTOT);
    for (i=0;i<NTOT;i++) {
        flmoon(i,nph,jd,frac);
        data[i]=jd;
    }
    avevar(data,ave,vrnce);
    cout << "Average = " << setw(12) << ave;
    cout << " Variance = " << setw(13) << vrnce << endl;
    return 0;
}
```

Be sure that the NR source code files are in a place that your compiler can find them to `#include`. Compile and run the above file. (We can't tell you how to do this part.) Output should be something like this:

```
Average =  2.41532e+06 Variance =       30480.7
```

2. *Yes, but where do I actually get the NR source code as computer files?*

You can buy a code subscription, or a one-time code download, at the Web site http://www.nr.com, or you can get the code on media published by Cambridge

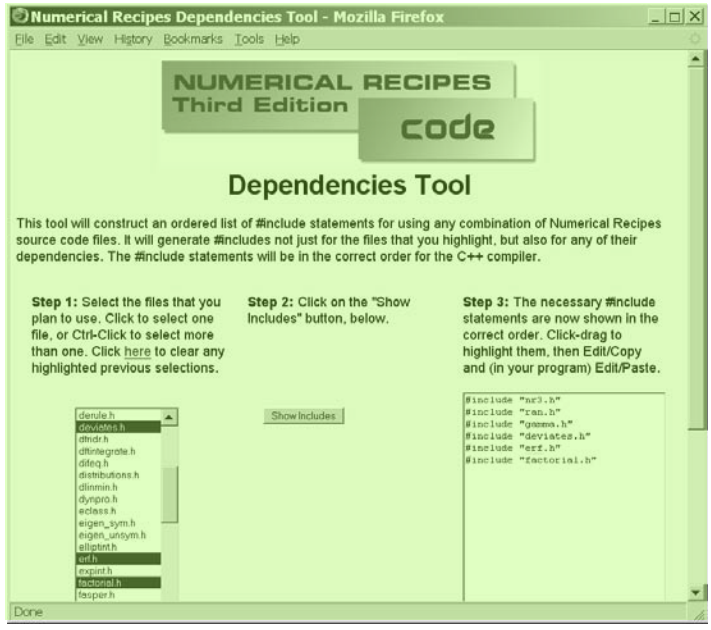

Figure 1.0.1. The interactive page located at http://www.nr.com/dependencies sorts out the dependencies for any combination of *Numerical Recipes* routines, giving an ordered list of the necessary #include files.

University Press (e.g., from Amazon.com or your favorite online or physical bookstore). The code comes with a personal, single-user license (see License and Legal Information on p. xix). The reason that the book (or its electronic version) and the code license are sold separately is to help keep down the price of each. Also, making these products separate meets the needs of more users: Your company or educational institution may have a site license — ask them.

3. *How do I know which files to* `#include`*? It's hard to sort out the dependencies among all the routines.*

In the margin next to each code listing is the name of the source code file that it is in. Make a list of the source code files that you are using. Then go to http://www.nr.com/dependencies and click on the name of each source code file. The interactive Web page will return a list of the necessary `#includes`, in the correct order, to satisfy all dependencies. Figure 1.0.1 will give you an idea of how this works.

4. *What is all this* `Doub`*,* `Int`*,* `VecDoub`*, etc., stuff?*

We always use defined types, not built-in types, so that they can be redefined if necessary. The definitions are in `nr3.h`. Generally, as you can guess, `Doub` means `double`, `Int` means `int`, and so forth. Our convention is to begin all defined types with an uppercase letter. `VecDoub` is a vector class type. Details on our types are in §1.4.

5. *What are* Numerical Recipes *Webnotes?*

Numerical Recipes Webnotes are documents, accessible on the Web, that include some code implementation listings, or other highly specialized topics, that are not included in the paper version of the book. A list of all Webnotes is at

Tested Operating Systems and Compilers	
O/S	Compiler
Microsoft Windows XP SP2	Visual C++ ver. 14.00 (Visual Studio 2005)
Microsoft Windows XP SP2	Visual C++ ver. 13.10 (Visual Studio 2003)
Microsoft Windows XP SP2	Intel C++ Compiler ver. 9.1
Novell SUSE Linux 10.1	GNU GCC (g++) ver. 4.1.0
Red Hat Enterprise Linux 4 (64-bit)	GNU GCC (g++) ver. 3.4.6 and ver. 4.1.0
Red Hat Linux 7.3	Intel C++ Compiler ver. 9.1
Apple Mac OS X 10.4 (Tiger) Intel Core	GNU GCC (g++) ver. 4.0.1

http://www.nr.com/webnotes. By moving some specialized material into Webnotes, we are able to keep down the size and price of the paper book. Webnotes are automatically included in the electronic version of the book; see next question.

6. *I am a post-paper person. I want* Numerical Recipes *on my laptop. Where do I get the complete, fully electronic version?*

A fully electronic version of *Numerical Recipes* is available by annual subscription. You can subscribe instead of, or in addition to, owning a paper copy of the book. A subscription is accessible via the Web, downloadable, printable, and, unlike any paper version, always up to date with the latest corrections. Since the electronic version does not share the page limits of the printed version, it will grow over time by the addition of completely new sections, available only electronically. This, we think, is the future of *Numerical Recipes* and perhaps of technical reference books generally. We anticipate various electronic formats, changing with time as technologies for display and rights management continuously improve: We place a big emphasis on user convenience and usability. See http://www.nr.com/electronic for further information.

7. *Are there bugs in NR?*

Of course! By now, most NR code has the benefit of long-time use by a large user community, but new bugs are sure to creep in. Look at http://www.nr.com for information about known bugs, or to report apparent new ones.

1.0.3 Computational Environment and Program Validation

The code in this book should run without modification on any compiler that implements the ANSI/ISO C++ standard, as described, for example, in Stroustrup's book [11].

As surrogates for the large number of hardware and software configurations, we have tested all the code in this book on the combinations of operating systems and compilers shown in the table above.

In validating the code, we have taken it directly from the machine-readable form of the book's manuscript, so that we have tested exactly what is printed. (This does not, of course, mean that the code is bug-free!)

1.0.4 About References 1

You will find references, and suggestions for further reading, listed at the end of most sections of this book. References are cited in the text by bracketed numbers like this [12]. 6

We do not pretend to any degree of bibliographical completeness in this book. For topics where a substantial secondary literature exists (discussion in textbooks, reviews, etc.) we often limit our references to a few of the more useful secondary sources, especially those with good references to the primary literature. Where the existing secondary literature is insufficient, we give references to a few primary sources that are intended to serve as starting points for further reading, not as complete bibliographies for the field. 1

Since progress is ongoing, it is inevitable that our references for many topics are already, or will soon become, out of date. We have tried to include older references that are good for "forward" Web searching: A search for more recent papers that cite the references given should lead you to the most current work. 3

Web references and URLs present a problem, because there is no way for us to guarantee that they will still be there when you look for them. A date like 2007+ means "it was there in 2007." We try to give citations that are complete enough for you to find the document by Web search, even if it has moved from the location listed. 4

The order in which references are listed is not necessarily significant. It reflects a compromise between listing cited references in the order cited, and listing suggestions for further reading in a roughly prioritized order, with the most useful ones first. 5

1.0.5 About "Advanced Topics" 2

Material set in smaller type, like this, signals an "advanced topic," either one outside of the main argument of the chapter, or else one requiring of you more than the usual assumed mathematical background, or else (in a few cases) a discussion that is more speculative or an algorithm that is less well tested. Nothing important will be lost if you skip the advanced topics on a first reading of the book. 2

Here is a function for getting the Julian Day Number from a calendar date. 7

calendar.h 10

```
Int julday(const Int mm, const Int id, const Int iyyy) {
```

In this routine julday returns the Julian Day Number that begins at noon of the calendar date specified by month mm, day id, and year iyyy, all integer variables. Positive year signifies A.D.; negative, B.C. Remember that the year after 1 B.C. was 1 A.D. 8

```
    const Int IGREG=15+31*(10+12*1582);        Gregorian Calendar adopted Oct. 15, 1582
    Int ja,jul,jy=iyyy,jm;                                                              9
    if (jy == 0) throw("julday: there is no year zero.");
    if (jy < 0) ++jy;
    if (mm > 2) {
        jm=mm+1;
    } else {
        --jy;
        jm=mm+13;
    }
    jul = Int(floor(365.25*jy)+floor(30.6001*jm)+id+1720995);
    if (id+31*(mm+12*iyyy) >= IGREG) {        Test whether to change to Gregorian Calendar.
        ja=Int(0.01*jy);
        jul += 2-ja+Int(0.25*ja);
    }
    return jul;
}
```

And here is its inverse.

```
void caldat(const Int julian, Int &mm, Int &id, Int &iyyy) {
```
Inverse of the function julday given above. Here julian is input as a Julian Day Number, and the routine outputs mm,id, and iyyy as the month, day, and year on which the specified Julian Day started at noon.

```
    const Int IGREG=2299161;
    Int ja,jalpha,jb,jc,jd,je;

    if (julian >= IGREG) {          Cross-over to Gregorian Calendar produces this correction.
        jalpha=Int((Doub(julian-1867216)-0.25)/36524.25);
        ja=julian+1+jalpha-Int(0.25*jalpha);
    } else if (julian < 0) {        Make day number positive by adding integer number of
        ja=julian+36525*(1-julian/36525);   Julian centuries, then subtract them off
    } else                                  at the end.
        ja=julian;
    jb=ja+1524;
    jc=Int(6680.0+(Doub(jb-2439870)-122.1)/365.25);
    jd=Int(365*jc+(0.25*jc));
    je=Int((jb-jd)/30.6001);
    id=jb-jd-Int(30.6001*je);
    mm=je-1;
    if (mm > 12) mm -= 12;
    iyyy=jc-4715;
    if (mm > 2) --iyyy;
    if (iyyy <= 0) --iyyy;
    if (julian < 0) iyyy -= 100*(1-julian/36525);
}
```

calendar.h

As an exercise, you might try using these functions, along with `flmoon` in §1.0, to search for future occurrences of a full moon on Friday the 13th. (Answers, in time zone GMT minus 5: 9/13/2019 and 8/13/2049.) For additional calendrical algorithms, applicable to various historical calendars, see [13].

CITED REFERENCES AND FURTHER READING:

Visual Numerics, 2007+, *IMSL Numerical Libraries*, at `http://www.vni.com`.[1]

Numerical Algorithms Group, 2007+, *NAG Numerical Library*, at `http://www.nag.co.uk`.[2]

Wolfram Research, Inc., 2007+, *Mathematica*, at `http://www.wolfram.com`.[3]

The MathWorks, Inc., 2007+, *MATLAB*, at `http://www.mathworks.com`.[4]

Maplesoft, Inc., 2007+, *Maple*, at `http://www.maplesoft.com`.[5]

GNU Scientific Library, 2007+, at `http://www.gnu.org/software/gsl`.[6]

Netlib Repository, 2007+, at `http://www.netlib.org`.[7]

Scilab Scientific Software Package, 2007+, at `http://www.scilab.org`.[8]

GNU Octave, 2007+, at `http://www.gnu.org/software/octave`.[9]

R Software Environment for Statistical Computing and Graphics, 2007+, at `http://www.r-project.org`.[10]

Stroustrup, B. 1997, *The C++ Programming Language*, 3rd ed. (Reading, MA: Addison-Wesley).[11]

Meeus, J. 1982, *Astronomical Formulae for Calculators*, 2nd ed., revised and enlarged (Richmond, VA: Willmann-Bell).[12]

Hatcher, D.A. 1984, "Simple Formulae for Julian Day Numbers and Calendar Dates," *Quarterly Journal of the Royal Astronomical Society*, vol. 25, pp. 53–55; see also *op. cit.* 1985, vol. 26, pp. 151–155, and 1986, vol. 27, pp. 506–507.[13]

1.1 Error, Accuracy, and Stability 1

Computers store numbers not with infinite precision but rather in some approximation that can be packed into a fixed number of *bits* (binary digits) or *bytes* (groups of 8 bits). Almost all computers allow the programmer a choice among several different such *representations* or *data types*. Data types can differ in the number of bits utilized (the *wordlength*), but also in the more fundamental respect of whether the stored number is represented in *fixed-point* (like `int`) or *floating-point* (like `float` or `double`) format.

A number in integer representation is exact. Arithmetic between numbers in integer representation is also exact, with the provisos that (i) the answer is not outside the range of (usually, signed) integers that can be represented, and (ii) that division is interpreted as producing an integer result, throwing away any integer remainder.

1.1.1 Floating-Point Representation 2

In a floating-point representation, a number is represented internally by a sign bit S (interpreted as plus or minus), an exact integer exponent E, and an exactly represented binary mantissa M. Taken together these represent the number

$$S \times M \times b^{E-e} \quad (1.1.1)$$

where b is the base of the representation ($b = 2$ almost always), and e is the *bias* of the exponent, a fixed integer constant for any given machine and representation.

	S	E	F	Value
`float`	any	1–254	any	$(-1)^S \times 2^{E-127} \times 1.F$
	any	0	nonzero	$(-1)^S \times 2^{-126} \times 0.F*$
	0	0	0	$+0.0$
	1	0	0	-0.0
	0	255	0	$+\infty$
	1	255	0	$-\infty$
	any	255	nonzero	`NaN`
`double`	any	1–2046	any	$(-1)^S \times 2^{E-1023} \times 1.F$
	any	0	nonzero	$(-1)^S \times 2^{-1022} \times 0.F*$
	0	0	0	$+0.0$
	1	0	0	-0.0
	0	2047	0	$+\infty$
	1	2047	0	$-\infty$
	any	2047	nonzero	`NaN`
*unnormalized values				

Several floating-point bit patterns can in principle represent the same number. If $b = 2$, for example, a mantissa with leading (high-order) zero bits can be left-shifted, i.e., multiplied by a power of 2, if the exponent is decreased by a compensating amount. Bit patterns that are "as left-shifted as they can be" are termed *normalized*.

Virtually all modern processors share the same floating-point data representations, namely those specified in IEEE Standard 754-1985 [1]. (For some discussion of nonstandard processors, see §22.2.) For 32-bit `float` values, the exponent is represented in 8 bits (with $e = 127$) the mantissa in 23; for 64-bit `double` values, the exponent is 11 bits (with $e = 1023$), the mantissa, 52. An additional trick is used for the mantissa for most nonzero floating values: Since the high-order bit of a properly normalized mantissa is *always* one, the stored mantissa bits are viewed as being preceded by a "phantom" bit with the value 1. In other words, the mantissa M has the numerical value $1.F$, where F (called the *fraction*) consists of the bits (23 or 52 in number) that are actually stored. This trick gains a little "bit" of precision.

Here are some examples of IEEE 754 representations of `double` values:

$$\begin{aligned}
&\texttt{0 01111111111 0000} \text{ (+ 48 more zeros)} = +1 \times 2^{1023-1023} \times 1.0_2 = 1. \\
&\texttt{1 01111111111 0000} \text{ (+ 48 more zeros)} = -1 \times 2^{1023-1023} \times 1.0_2 = -1. \\
&\texttt{0 01111111111 1000} \text{ (+ 48 more zeros)} = +1 \times 2^{1023-1023} \times 1.1_2 = 1.5 \\
&\texttt{0 10000000000 0000} \text{ (+ 48 more zeros)} = +1 \times 2^{1024-1023} \times 1.0_2 = 2. \\
&\texttt{0 10000000001 1010} \text{ (+ 48 more zeros)} = +1 \times 2^{1025-1023} \times 1.1010_2 = 6.5
\end{aligned} \tag{1.1.2}$$

You can examine the representation of any value by code like this:

```
union Udoub {
    double d;
    unsigned char c[8];
};

void main() {
    Udoub u;
    u.d = 6.5;
    for (int i=7;i>=0;i--) printf("%02x",u.c[i]);
    printf("\n");
}
```

This is C, and deprecated style, but it will work. On most processors, including Intel Pentium and successors, you'll get the printed result `401a000000000000`, which (writing out each hex digit as four binary digits) is the last line in equation (1.1.2). If you get the bytes (groups of two hex digits) in reverse order, then your processor is *big-endian* instead of *little-endian*: The IEEE 754 standard does not specify (or care) in which order the bytes in a floating-point value are stored.

The IEEE 754 standard includes representations of positive and negative infinity, positive and negative zero (treated as computationally equivalent, of course), and also `NaN` ("not a number"). The table on the previous page gives details of how these are represented.

The reason for representing some *unnormalized* values, as shown in the table, is to make "underflow to zero" more graceful. For a sequence of smaller and smaller values, after you pass the smallest normalizable value (with magnitude 2^{-127} or 2^{-1023}, see table), you start right-shifting the leading bit of the mantissa. Although

you gradually lose precision, you don't actually underflow to zero until 23 or 52 bits later.

When a routine needs to know properties of the floating-point representation, it can reference the `numeric_limits` class, which is part of the C++ Standard Library. For example, `numeric_limits<double>::min()` returns the smallest normalized `double` value, usually $2^{-1022} \approx 2.23 \times 10^{-308}$. For more on this, see §22.2.

1.1.2 Roundoff Error

Arithmetic among numbers in floating-point representation is not exact, even if the operands happen to be exactly represented (i.e., have exact values in the form of equation 1.1.1). For example, two floating numbers are added by first right-shifting (dividing by two) the mantissa of the smaller (in magnitude) one and simultaneously increasing its exponent until the two operands have the same exponent. Low-order (least significant) bits of the smaller operand are lost by this shifting. If the two operands differ too greatly in magnitude, then the smaller operand is effectively replaced by zero, since it is right-shifted to oblivion.

The smallest (in magnitude) floating-point number that, when added to the floating-point number 1.0, produces a floating-point result different from 1.0 is termed the *machine accuracy* ϵ_m. IEEE 754 standard `float` has ϵ_m about 1.19×10^{-7}, while `double` has about 2.22×10^{-16}. Values like this are accessible as, e.g., `numeric _limits <double>::epsilon()`. (A more detailed discussion of machine characteristics is in §22.2.) Roughly speaking, the machine accuracy ϵ_m is the fractional accuracy to which floating-point numbers are represented, corresponding to a change of one in the least significant bit of the mantissa. Pretty much any arithmetic operation among floating numbers should be thought of as introducing an additional fractional error of at least ϵ_m. This type of error is called *roundoff error*.

It is important to understand that ϵ_m is not the smallest floating-point number that can be represented on a machine. *That* number depends on how many bits there are in the exponent, while ϵ_m depends on how many bits there are in the mantissa.

Roundoff errors accumulate with increasing amounts of calculation. If, in the course of obtaining a calculated value, you perform N such arithmetic operations, you *might* be so lucky as to have a total roundoff error on the order of $\sqrt{N}\epsilon_m$, if the roundoff errors come in randomly up or down. (The square root comes from a random-walk.) However, this estimate can be very badly off the mark for two reasons:

(1) It very frequently happens that the regularities of your calculation, or the peculiarities of your computer, cause the roundoff errors to accumulate preferentially in one direction. In this case the total will be of order $N\epsilon_m$.

(2) Some especially unfavorable occurrences can vastly increase the roundoff error of single operations. Generally these can be traced to the subtraction of two very nearly equal numbers, giving a result whose only significant bits are those (few) low-order ones in which the operands differed. You might think that such a "coincidental" subtraction is unlikely to occur. Not always so. Some mathematical expressions magnify its probability of occurrence tremendously. For example, in the familiar formula for the solution of a quadratic equation,

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{1.1.3}$$

the addition becomes delicate and roundoff-prone whenever $b > 0$ and $|ac| \ll b^2$. (In §5.6 we will learn how to avoid the problem in this particular case.)

1.1.3 Truncation Error

Roundoff error is a characteristic of computer hardware. There is another, different, kind of error that is a characteristic of the program or algorithm used, independent of the hardware on which the program is executed. Many numerical algorithms compute "discrete" approximations to some desired "continuous" quantity. For example, an integral is evaluated numerically by computing a function at a discrete set of points, rather than at "every" point. Or, a function may be evaluated by summing a finite number of leading terms in its infinite series, rather than all infinity terms. In cases like this, there is an adjustable parameter, e.g., the number of points or of terms, such that the "true" answer is obtained only when that parameter goes to infinity. Any practical calculation is done with a finite, but sufficiently large, choice of that parameter.

The discrepancy between the true answer and the answer obtained in a practical calculation is called the *truncation error*. Truncation error would persist even on a hypothetical, "perfect" computer that had an infinitely accurate representation and no roundoff error. As a general rule there is not much that a programmer can do about roundoff error, other than to choose algorithms that do not magnify it unnecessarily (see discussion of "stability" below). Truncation error, on the other hand, is entirely under the programmer's control. In fact, it is only a slight exaggeration to say that clever minimization of truncation error is practically the entire content of the field of numerical analysis!

Most of the time, truncation error and roundoff error do not strongly interact with one another. A calculation can be imagined as having, first, the truncation error that it would have if run on an infinite-precision computer, "plus" the roundoff error associated with the number of operations performed.

1.1.4 Stability

Sometimes an otherwise attractive method can be *unstable*. This means that any roundoff error that becomes "mixed into" the calculation at an early stage is successively magnified until it comes to swamp the true answer. An unstable method would be useful on a hypothetical, perfect computer; but in this imperfect world it is necessary for us to require that algorithms be stable — or if unstable that we use them with great caution.

Here is a simple, if somewhat artificial, example of an unstable algorithm: Suppose that it is desired to calculate all integer powers of the so-called "Golden Mean," the number given by

$$\phi \equiv \frac{\sqrt{5}-1}{2} \approx 0.61803398 \tag{1.1.4}$$

It turns out (you can easily verify) that the powers ϕ^n satisfy a simple recursion relation,

$$\phi^{n+1} = \phi^{n-1} - \phi^n \tag{1.1.5}$$

Thus, knowing the first two values $\phi^0 = 1$ and $\phi^1 = 0.61803398$, we can successively apply (1.1.5) performing only a single subtraction, rather than a slower

multiplication by ϕ, at each stage.

Unfortunately, the recurrence (1.1.5) also has *another* solution, namely the value $-\frac{1}{2}(\sqrt{5}+1)$. Since the recurrence is linear, and since this undesired solution has magnitude greater than unity, any small admixture of it introduced by roundoff errors will grow exponentially. On a typical machine, using a 32-bit `float`, (1.1.5) starts to give completely wrong answers by about $n = 16$, at which point ϕ^n is down to only 10^{-4}. The recurrence (1.1.5) is *unstable* and cannot be used for the purpose stated.

We will encounter the question of stability in many more sophisticated guises later in this book.

CITED REFERENCES AND FURTHER READING:

IEEE, 1985, *ANSI/IEEE Std 754–1985: IEEE Standard for Binary Floating-Point Numbers* (New York: IEEE).[1]

Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), Chapter 1.

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 2.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §1.3.

Wilkinson, J.H. 1964, *Rounding Errors in Algebraic Processes* (Englewood Cliffs, NJ: Prentice-Hall).

1.2 C Family Syntax

Not only C++, but also Java, C#, and (to varying degrees) other computer languages, share a lot of small-scale syntax with the older C language [1]. By small scale, we mean operations on built-in types, simple expressions, control structures, and the like. In this section, we review some of the basics, give some hints on good programming, and mention some of our conventions and habits.

1.2.1 Operators

A first piece of advice might seem superfluous if it were not so often ignored: You should learn all the C operators and their precedence and associativity rules. You might not yourself want to write

```
n << 1 | 1
```

as a synonym for `2*n+1` (for positive integer `n`), but you definitely do need to be able to see at a glance that

```
n << 1 + 1
```

is not at all the same thing! Please study the table on the next page while you brush your teeth every night. While the occasional set of unnecessary parentheses, for clarity, is hardly a sin, code that is habitually overparenthesized is annoying and hard to read.

Operator Precedence and Associativity Rules in C and C++ 1		
`::`	scope resolution	left-to-right
`()`	function call	left-to-right
`[]`	array element (subscripting)	
`.`	member selection	
`->`	member selection (by pointer)	
`++`	post increment	**right-to-left**
`--`	post decrement	
`!`	logical not	**right-to-left**
`~`	bitwise complement	
`-`	unary minus	
`++`	pre increment	
`--`	pre decrement	
`&`	address of	
`*`	contents of (dereference)	
`new`	create	
`delete`	destroy	
`(type)`	cast to `type`	
`sizeof`	size in bytes	
`*`	multiply	left-to-right
`/`	divide	
`%`	remainder	
`+`	add	left-to-right
`-`	subtract	
`<<`	bitwise left shift	left-to-right
`>>`	bitwise right shift	
`<`	arithmetic less than	left-to-right
`>`	arithmetic greater than	
`<=`	arithmetic less than or equal to	
`>=`	arithmetic greater than or equal to	
`==`	arithmetic equal	left-to-right
`!=`	arithmetic not equal	
`&`	bitwise and	left-to-right
`^`	bitwise exclusive or	left-to-right
`|`	bitwise or	left-to-right
`&&`	logical and	left-to-right
`||`	logical or	left-to-right
`?  :`	conditional expression	**right-to-left**
`=`	assignment operator	**right-to-left**
also `+= -= *= /= %= <<= >>= &= ^= |=`		
`,`	sequential expression	left-to-right

1.2.2 Control Structures

These should all be familiar to you.

Iteration. In C family languages simple iteration is performed with a `for` loop, for example

```
for (j=2;j<=1000;j++) {
    b[j]=a[j-1];
    a[j-1]=j;
}
```

It is conventional to indent the block of code that is acted upon by the control structure, leaving the structure itself unindented. We like to put the initial curly brace on the same line as the `for` statement, instead of on the next line. This saves a full line of white space, and our publisher loves us for it.

Conditional. The conditional or `if` structure looks, in full generality, like this:

```
if (...) {
    ...
}
else if (...) {
    ...
}
else {
    ...
}
```

However, since compound-statement curly braces are required only when there is more than one statement in a block, the `if` construction can be somewhat less explicit than that shown above. Some care must be exercised in constructing nested `if` clauses. For example, consider the following:

```
if (b > 3)
    if (a > 3) b += 1;
else b -= 1;                /* questionable! */
```

As judged by the indentation used on successive lines, the intent of the writer of this code is the following: 'If b is greater than 3 and a is greater than 3, then increment b. If b is not greater than 3, then decrement b.' According to the rules, however, the actual meaning is 'If b is greater than 3, then evaluate a. If a is greater than 3, then increment b, and if a is less than or equal to 3, decrement b.' The point is that an `else` clause is associated with the most recent open `if` statement, no matter how you lay it out on the page. Such confusions in meaning are easily resolved by the inclusion of braces that clarify your intent and improve the program. The above fragment should be written as

```
if (b > 3) {
    if (a > 3) b += 1;
} else {
    b -= 1;
}
```

While iteration. Alternative to the `for` iteration is the `while` structure, for example,

```
while (n < 1000) {
    n *= 2;
    j += 1;
}
```

The control clause (in this case `n < 1000`) is evaluated before each iteration. If the clause is not true, the enclosed statements will not be executed. In particular, if this code is encountered at a time when `n` is greater than or equal to 1000, the statements will not even be executed once.

Do-While iteration. Companion to the `while` iteration is a related control structure that tests its control clause at the *end* of each iteration:

```
do {
    n *= 2;
    j += 1;
} while (n < 1000);
```

In this case, the enclosed statements will be executed at least once, independent of the initial value of `n`.

Break and Continue. You use the `break` statement when you have a loop that is to be repeated indefinitely until some condition *tested somewhere in the middle of the loop* (and possibly tested in more than one place) becomes true. At that point you wish to exit the loop and proceed with what comes after it. In C family languages the simple `break` statement terminates execution of the innermost `for`, `while`, `do`, or `switch` construction and proceeds to the next sequential instruction. A typical usage might be

```
for(;;) {
    ...                          (statements before the test)
    if (...) break;
    ...                          (statements after the test)
}
...                              (next sequential instruction)
```

Companion to `break` is `continue`, which transfers program control to the end of the body of the smallest enclosing `for`, `while`, or `do` statement, but *just inside* that body's terminating curly brace. In general, this results in the execution of the next loop test associated with that body.

1.2.3 How Tricky Is Too Tricky?

Every programmer is occasionally tempted to write a line or two of code that is so elegantly tricky that all who read it will stand in awe of its author's intelligence. Poetic justice is that it is usually that same programmer who gets stumped, later on, trying to understand his or her own creation. You might momentarily be proud of yourself at writing the single line

```
k=(2-j)*(1+3*j)/2;
```

if you want to permute cyclically one of the values $j = (0, 1, 2)$ into respectively $k = (1, 2, 0)$. You will regret it later, however. Better, and likely also faster, is

```
k=j+1;
if (k == 3) k=0;
```
21

On the other hand, it can also be a mistake, or at least suboptimal, to be too ploddingly literal, as in 4

```
switch (j) {
    case 0: k=1; break;
    case 1: k=2; break;
    case 2: k=0; break;
    default: {
        cerr << "unexpected value for j";
        exit(1);
    }
}
```
17

This (or similar) might be the house style if you are one of 10^5 programmers working for a megacorporation, but if you are programming for your own research, or within a small group of collaborators, this kind of style will soon cause you to lose the forest for the trees. You need to find the right personal balance between obscure trickery and boring prolixity. A good rule is that you should always write code that is *slightly less tricky than you are willing to read, but only slightly.* 1

There is a fine line between being tricky (bad) and being idiomatic (good). *Idioms* are short expressions that are sufficiently common, or sufficiently self-explanatory, that you can use them freely. For example, testing an integer `n`'s even- or odd-ness by 2

```
if (n & 1) ...
```
16

is, we think, much preferable to 9

```
if (n % 2 == 1) ...
```
14

We similarly like to double a positive integer by writing 6

```
n <<= 1;
```
3

or construct a mask of `n` bits by writing 5

```
(1 << n) - 1
```
18

and so forth. 8

Some idioms are worthy of consideration even when they are not so immediately obvious. S.E. Anderson [2] has collected a number of "bit-twiddling hacks," of which we show three here: 3

The test 12

```
if ((v&(v-1))==0) {}
```
Is a power of 2 or zero. 19

tests whether `v` is a power of 2. If you care about the case $v = 0$, you have to write 7

```
if (v&&((v&(v-1))==0)) {}
```
Is a power of 2. 13

The idiom 10

```
for (c=0;v;c++) v &= v - 1;
```
1

gives as `c` the number of set ($= 1$) bits in a positive or unsigned integer `v` (destroying `v` in the process). The number of iterations is only as many as the number of bits set. 11

The idiom 15

```
v--;
v |= v >> 1; v |= v >> 2; v |= v >> 4; v |= v >> 8; v |= v >> 16;
v++;
```
20

rounds a positive (or unsigned) 32-bit integer v up to the next power of 2 that is $\geq$ v.

When we use the bit-twiddling hacks, we'll include an explanatory comment in the code.

1.2.4 Utility Macros or Templated Functions

The file nr3.h includes, among other things, definitions for the functions

```
MAX(a,b)
MIN(a,b)
SWAP(a,b)
SIGN(a,b)
```

These are all self-explanatory, except possibly the last. SIGN(a,b) returns a value with the same magnitude as a and the same sign as b. These functions are all implemented as templated inline functions, so that they can be used for all argument types that make sense semantically. Implementation as macros is also possible.

CITED REFERENCES AND FURTHER READING:

Harbison, S.P., and Steele, G.L., Jr. 2002, *C: A Reference Manual*, 5th ed. (Englewood Cliffs, NJ: Prentice-Hall).[1]

Anderson, S.E. 2006, "Bit Twiddling Hacks," at http://graphics.stanford.edu/~seander/bithacks.html.[2]

1.3 Objects, Classes, and Inheritance

An *object* or *class* (the terms are interchangeable) is a program structure that groups together some variables, or functions, or both, in such a way that all the included variables or functions "see" each other and can interact intimately, while most of this internal structure is hidden from other program structures and units. Objects make possible *object-oriented programming* (OOP), which has become recognized as the almost unique successful paradigm for creating complex software. The key insight in OOP is that objects have *state* and *behavior*. The state of the object is described by the values stored in its member variables, while the possible behavior is determined by the member functions. We will use objects in other ways as well.

The terminology surrounding OOP can be confusing. Objects, classes, and structures pretty much refer to the same thing. Member functions in a class are often referred to as *methods* belonging to that class. In C++, objects are defined with either the keyword class or the keyword struct. These differ, however, in the details of how rigorously they hide the object's internals from public view. Specifically,

```
struct SomeName { ...
```

is defined as being the same as

```
class SomeName {
public: ...
```

In this book we *always* use struct. This is not because we deprecate the use of public and private access specifiers in OOP, but only because such access control would add little to understanding the underlying numerical methods that are the focus of this book. In fact, access specifiers could impede your understanding, because

you would be constantly moving things from private to public (and back again) as you program different test cases and want to examine different internal, normally private, variables.

Because our classes are declared by `struct`, not `class`, use of the word "class" is potentially confusing, and we will usually try to avoid it. So "object" means `struct`, which is really a class!

If you are an OOP beginner, it is important to understand the distinction between defining an object and instantiating it. You define an object by writing code like this:

```
struct Twovar {
    Doub a,b;
    Twovar(const Doub aa, const Doub bb) : a(aa), b(bb) {}
    Doub sum() {return a+b;}
    Doub diff() {return a-b;}
};
```

This code does not create a `Twovar` object. It only tells the compiler how to create one when, later in your program, you tell it to do so, for example by a declaration like,

```
Twovar mytwovar(3.,5.);
```

which invokes the `Twovar` constructor and creates an instance of (or *instantiates*) a `Twovar`. In this example, the constructor also sets the internal variables `a` and `b` to 3 and 5, respectively. You can have any number of simultaneously existing, noninteracting, instances:

```
Twovar anothertwovar(4.,6.);
Twovar athirdtwovar(7.,8.);
```

We have already promised you that this book is not a textbook in OOP, or the C++ language; so we will go no farther here. If you need more, good references are [1-4].

1.3.1 Simple Uses of Objects

We use objects in various ways, ranging from trivial to quite complex, depending on the needs of the specific numerical method that is being discussed. As mentioned in §1.0, this lack of consistency means that *Numerical Recipes* is not a useful examplar of a program library (or, in an OOP context, a *class library*). It also means that, somewhere in this book, you can probably find an example of every possible way to think about objects in numerical computing! (We hope that you will find this a plus.)

Object for Grouping Functions. Sometimes an object just collects together a group of closely related functions, not too differently from the way that you might use a `namespace`. For example, a simplification of Chapter 6's object `Erf` looks like:

```
struct Erf {
    Doub erf(Doub x);
    Doub erfc(Doub x);
    Doub inverf(Doub p);
    Doub inverfc(Doub p);
    Doub erfccheb(Doub z);
};
```

No constructor needed.

As will be explained in §6.2, the first four methods are the ones intended to be called by the user, giving the error function, complementary error function, and the two

corresponding inverse functions. But these methods share some code and also use common code in the last method, `erfccheb`, which the user will normally ignore completely. It therefore makes sense to group the whole collection as an `Erf` object. About the only disadvantage of this is that you must instantiate an `Erf` object before you can use (say) the `erf` function:

```
Erf myerf;                          The name myerf is arbitrary.
...
Doub y = myerf.erf(3.);
```

Instantiating the object doesn't actually *do* anything here, because `Erf` contains no variables (i.e., has no stored state). It just tells the compiler what local name you are going to use in referring to its member functions. (We would normally use the name `erf` for the instance of `Erf`, but we thought that `erf.erf(3.)` would be confusing in the above example.)

Object for Standardizing an Interface. In §6.14 we'll discuss a number of useful standard probability distributions, for example, normal, Cauchy, binomial, Poisson, etc. Each gets its own object definition, for example,

```
struct Cauchydist {
    Doub mu, sig;
    Cauchydist(Doub mmu = 0., Doub ssig = 1.) : mu(mmu), sig(ssig) {}
    Doub p(Doub x);
    Doub cdf(Doub x);
    Doub invcdf(Doub p);
};
```

where the function `p` returns the probability density, the function `cdf` returns the cumulative distribution function (cdf), and the function `invcdf` returns the inverse of the cdf. Because the interface is consistent across all the different probability distributions, you can change which distribution a program is using by changing a single program line, for example from

```
Cauchydist mydist();
```

to

```
Normaldist mydist();
```

All subsequent references to functions like `mydist.p`, `mydist.cdf`, and so on, are thus changed automatically. This is hardly OOP at all, but it can be very convenient.

Object for Returning Multiple Values. It often happens that a function computes more than one useful quantity, but you don't know which one or ones the user is actually interested in on that particular function call. A convenient use of objects is to save all the potentially useful results and then let the user grab those that are of interest. For example, a simplified version of the `Fitab` structure in Chapter 15, which fits a straight line $y = a + bx$ to a set of data points `xx` and `yy`, looks like this:

```
struct Fitab {
    Doub a, b;
    Fitab(const VecDoub &xx, const VecDoub &yy);          Constructor.
};
```

(We'll discuss `VecDoub` and related matters below, in §1.4.) The user calculates the fit by calling the constructor with the data points as arguments,

```
Fitab myfit(xx,yy);
```

Then the two "answers" a and b are separately available as `myfit.a` and `myfit.b`. We will see more elaborate examples throughout the book.

Objects That Save Internal State for Multiple Uses. This is classic OOP, worthy of the name. A good example is Chapter 2's `LUdcmp` object, which (in abbreviated form) looks like this:

```
struct LUdcmp {
    Int n;
    MatDoub lu;
    LUdcmp(const MatDoub &a);  Constructor.
    void solve(const VecDoub &b, VecDoub &x);
    void inverse(MatDoub &ainv);
    Doub det();
};
```

This object is used to solve linear equations and/or invert a matrix. You use it by creating an instance with your matrix `a` as the argument in the constructor. The constructor then computes and stores, in the internal matrix `lu`, a so-called LU decomposition of your matrix (see §2.3). Normally you won't use the matrix `lu` directly (though you could if you wanted to). Rather, you now have available the methods `solve()`, which returns a solution vector `x` for any right-hand side `b`, `inverse()`, which returns the inverse matrix, and `det()`, which returns the determinant of your matrix.

You can call any or all of `LUdcmp`'s methods in any order; you might well want to call `solve` multiple times, with different right-hand sides. If you have more than one matrix in your problem, you create a separate instance of `LUdcmp` for each one, for example,

```
LUdcmp alu(a), aalu(aa);
```

after which `alu.solve()` and `aalu.solve()` are the methods for solving linear equations for each respective matrix, `a` and `aa`; `alu.det()` and `aalu.det()` return the two determinants; and so forth.

We are not finished listing ways to use objects: Several more are discussed in the next few sections.

1.3.2 Scope Rules and Object Destruction

This last example, `LUdcmp`, raises the important issue of how to manage an object's time and memory usage within your program.

For a large matrix, the `LUdcmp` constructor does a lot of computation. You choose exactly where in your program you want this to occur in the obvious way, by putting the declaration

```
LUdcmp alu(a);
```

in just that place. The important distinction between a non-OOP language (like C) and an OOP language (like C++) is that, in the latter, declarations are not passive instructions to the compiler, but executable statments at run-time.

The `LUdcmp` constructor also, for a large matrix, grabs a lot of memory, to store the matrix `lu`. How do you take charge of this? That is, how do you communicate that it should save this state for as long as you might need it for calls to methods like `alu.solve()`, but not indefinitely?

The answer lies in C++'s strict and predictable rules about *scope*. You can start a temporary scope at any point by writing an open bracket, "{". You end that scope by a matching close bracket, "}". You can nest scopes in the obvious way. Any objects that are declared within a scope are destroyed (and their memory resources returned) when the end of the scope is reached. An example might look like this:

```
MatDoub a(1000,1000);          Create a big matrix,
VecDoub b(1000),x(1000);       and a couple of vectors.
...
{                              Begin temporary scope.
    LUdcmp alu(a);             Create object alu.
    ...
    alu.solve(b,x);            Use alu.
    ...
}                              End temporary scope. Resources in alu are freed.
...
Doub d = alu.det();            ERROR! alu is out of scope.
```

This example presumes that you have some other use for the matrix a later on. If not, then the the declaration of a should itself probably be inside the temporary scope.

Be aware that *all* program blocks delineated by braces are scope units. This includes the main block associated with a function definition and also blocks associated with control structures. In code like this,

```
for (;;) {
    ...
    LUdcmp alu(a);
    ...
}
```

a new instance of alu is created at each iteration and then destroyed at the end of that iteration. This might sometimes be what you intend (if the matrix a changes on each iteration, for example); but you should be careful not to let it happen unintentionally.

1.3.3 Functions and Functors

Many routines in this book take functions as input. For example, the quadrature (integration) routines in Chapter 4 take as input the function $f(x)$ to be integrated. For a simple case like $f(x) = x^2$, you code such a function simply as

```
Doub f(const Doub x) {
    return x*x;
}
```

and pass f as an argument to the routine. However, it is often useful to use a more general object to communicate the function to the routine. For example, $f(x)$ may depend on other variables or parameters that need to be communicated from the calling program. Or the computation of $f(x)$ may be associated with other subcalculations or information from other parts of the program. In non-OOP programming, this communication is usually accomplished with global variables that pass the information "over the head" of the routine that receives the function argument f.

C++ provides a better and more elegant solution: *function objects* or *functors*. A functor is simply an object in which the operator () has been overloaded to play the role of returning a function value. (There is no relation between this use of the word functor and its different meaning in pure mathematics.) The case $f(x) = x^2$ would now be coded as

```
struct Square {
    Doub operator()(const Doub x) {
        return x*x;
    }
};
```

To use this with a quadrature or other routine, you declare an instance of `Square`

```
Square g;
```

and pass `g` to the routine. Inside the quadrature routine, an invocation of `g(x)` returns the function value in the usual way.

In the above example, there's no point in using a functor instead of a plain function. But suppose you have a parameter in the problem, for example, $f(x) = cx^p$, where c and p are to be communicated from somewhere else in your program. You can set the parameters via a constructor:

```
struct Contimespow {
    Doub c,p;
    Contimespow(const Doub cc, const Doub pp) : c(cc), p(pp) {}
    Doub operator()(const Doub x) {
        return c*pow(x,p);
    }
};
```

In the calling program, you might declare the instance of `Contimespow` by

```
Contimespow h(4.,0.5);          Communicate c and p to the functor.
```

and later pass `h` to the routine. Clearly you can make the functor much more complicated. For example, it can contain other helper functions to aid in the calculation of the function value.

So should we implement all our routines to accept only functors and not functions? Luckily, we don't have to decide. We can write the routines so they can accept *either* a function or a functor. A routine accepting only a function to be integrated from a to b might be declared as

```
Doub someQuadrature(Doub func(const Doub), const Doub a, const Doub b);
```

To allow it to accept either functions or functors, we instead make it a *templated* function:

```
template <class T>
Doub someQuadrature(T &func, const Doub a, const Doub b);
```

Now the compiler figures out whether you are calling `someQuadrature` with a function or a functor and generates the appropriate code. If you call the routine in one place in your program with a function and in another with a functor, the compiler will handle that too.

We will use this capability to pass functors as arguments in many different places in the book where function arguments are required. There is a tremendous gain in flexibility and ease of use.

As a convention, when we write `Ftor`, we mean a functor like `Square` or `Contimespow` above; when we write `fbare`, we mean a "bare" function like `f` above; and when we write `ftor` (all in lower case), we mean an instantiation of a functor, that is, something declared like

```
Ftor ftor(...);          Replace the dots by your parameters, if any.
```

Of course your names for functors and their instantiations will be different.

Slightly more complicated syntax is involved in passing a function to an *object*

that is templated to accept either a function or functor. So if the object is

```
template <class T>
struct SomeStruct {
    SomeStruct(T &func, ...);  constructor
    ...
```

we would instantiate it with a functor like this:

```
Ftor ftor;
SomeStruct<Ftor> s(ftor, ...
```

but with a function like this:

```
SomeStruct<Doub (const Doub)> s(fbare, ...
```

In this example, `fbare` takes a single `const Doub` argument and returns a `Doub`. You must use the arguments and return type for your specific case, of course.

1.3.4 Inheritance

Objects can be defined as deriving from other, already defined, objects. In such *inheritance*, the "parent" class is called a *base class*, while the "child" class is called a *derived class*. A derived class has all the methods and stored state of its base class, plus it can add any new ones.

"Is-a" Relationships. The most straightforward use of inheritance is to describe so-called *is-a* relationships. OOP texts are full of examples where the base class is `ZooAnimal` and a derived class is `Lion`. In other words, `Lion` "is-a" `ZooAnimal`. The base class has methods common to all `ZooAnimals`, for example `eat()` and `sleep()`, while the derived class extends the base class with additional methods specific to `Lion`, for example `roar()` and `eat_visitor()`.

In this book we use is-a inheritance less often than you might expect. Except in some highly stylized situations, like optimized matrix classes ("triangular matrix is-a matrix"), we find that the diversity of tasks in scientific computing does not lend itself to strict is-a hierarchies. There are exceptions, however. For example, in Chapter 7, we define an object `Ran` with methods for returning uniform random deviates of various types (e.g., `Int` or `Doub`). Later in the chapter, we define objects for returning other kinds of random deviates, for example normal or binomial. These are defined as derived classes of `Ran`, for example,

```
struct Binomialdev : Ran {};
```

so that they can share the machinery already in `Ran`. This is a true is-a relationship, because "binomial deviate is-a random deviate."

Another example occurs in Chapter 13, where objects `Daub4`, `Daub4i`, and `Daubs` are all derived from the `Wavelet` base class. Here `Wavelet` is an *abstract base class* or *ABC* [1,4] that has no content of its own. Rather, it merely specifies interfaces for all the methods that any `Wavelet` is required to have. The relationship is nevertheless is-a: "`Daub4` is-a `Wavelet`".

"Prerequisite" Relationships. Not for any dogmatic reason, but simply because it is convenient, we frequently use inheritance to pass on to an object a set of methods that it needs as prerequisites. This is especially true when the same set of prerequisites is used by more than one object. In this use of inheritance, the base class has no particular `ZooAnimal` unity; it may be a grab-bag. There is not a logical is-a relationship between the base and derived classes.

An example in Chapter 10 is `Bracketmethod`, which is a base class for several

minimization routines, but which simply provides a common method for the initial bracketing of a minimum. In Chapter 7, the `Hashtable` object provides prerequisite methods to its derived classes `Hash` and `Mhash`, but one cannot say, "`Mhash` is-a `Hashtable`" in any meaningful way. An extreme example, in Chapter 6, is the base class `Gauleg18`, which does nothing except provide a bunch of constants for Gauss-Legendre integration to derived classes `Beta` and `Gamma`, both of which need them. Similarly, long lists of constants are provided to the routines `StepperDopr853` and `StepperRoss` in Chapter 17 by base classes to avoid cluttering the coding of the algorithms.

Partial Abstraction. Inheritance can be used in more complicated or situation-specific ways. For example, consider Chapter 4, where elementary quadrature rules such as `Trapzd` and `Midpnt` are used as building blocks to construct more elaborate quadrature algorithms. The key feature these simple rules share is a mechanism for adding more points to an existing approximation to an integral to get the "next" stage of refinement. This suggests deriving these objects from an abstract base clase called `Quadrature`, which specifies that all objects derived from it must have a `next()` method. This is not a complete specification of a common is-a interface; it abstracts only one feature that turns out to be useful.

For example, in §4.6, the `Stiel` object invokes, in different situations, two different quadrature objects, `Trapzd` and `DErule`. These are not interchangeable. They have different constructor arguments and could not easily both be made `ZooAnimals` (as it were). `Stiel` of course knows about their differences. However, one of `Stiel`'s methods, `quad()`, doesn't (and shouldn't) know about these differences. It uses only the method `next()`, which exists, with different definitions, in both `Trapzd` and `DErule`.

While there are several different ways to deal with situations like this, an easy one is available once `Trapzd` and `DErule` have been given a common abstract base class `Quadrature` that contains nothing except a virtual interface to `next`. In a case like this, the base class is a minor design feature as far as the implementation of `Stiel` is concerned, almost an afterthought, rather than being the apex of a top-down design. As long as the usage is clear, there is nothing wrong with this.

Chapter 17, which discusses ordinary differential equations, has some even more complicated examples that combine inheritance and templating. We defer further discussion to there.

CITED REFERENCES AND FURTHER READING:

Stroustrup, B. 1997, *The C++ Programming Language*, 3rd ed. (Reading, MA: Addison-Wesley).[1]

Lippman, S.B., Lajoie, J., and Moo, B.E. 2005, *C++ Primer*, 4th ed. (Boston: Addison-Wesley).[2]

Keogh, J., and Giannini, M. 2004, *OOP Demystified* (Emeryville, CA: McGraw-Hill/Osborne).[3]

Cline, M., Lomow, G., and Girou, M. 1999, *C++ FAQs*, 2nd ed. (Boston: Addison-Wesley).[4]

1.4 Vector and Matrix Objects

The C++ Standard Library [1] includes a perfectly good `vector<>` template class. About the only criticism that one can make of it is that it is so feature-rich

that some compiler vendors neglect to squeeze the last little bit of performance out of its most elementary operations, for example returning an element by its subscript. That performance is extremely important in scientific applications; its occasional absence in C++ compilers is a main reason that many scientists still (as we write) program in C, or even in Fortran!

Also included in the C++ Standard Library is the class `valarray<>`. At one time, this was supposed to be a vector-like class that was optimized for numerical computation, including some features associated with matrices and multidimensional arrays. However, as reported by one participant,

> The valarray classes were not designed very well. In fact, nobody tried to determine whether the final specification worked. This happened because nobody felt "responsible" for these classes. The people who introduced valarrays to the C++ standard library left the committee a long time before the standard was finished. [1]

The result of this history is that C++, at least now, has a good (but not always reliably optimized) class for vectors and no dependable class at all for matrices or higher-dimensional arrays. What to do? We will adopt a strategy that emphasizes flexibility and assumes only a minimal set of properties for vectors and matrices. We will then provide our own, basic, classes for vectors and matrices. For most compilers, these are at least as efficient as `vector<>` and other vector and matrix classes in common use. But if, for you, they're not, then it is easy to change to a different set of classes, as we will explain.

1.4.1 Typedefs

Flexibility is achieved by having several layers of `typedef` type-indirection, resolved at compile time so that there is no run-time performance penalty. The first level of type-indirection, not just for vectors and matrices but for virtually all variables, is that we use user-defined type names instead of C++ fundamental types. These are defined in `nr3.h`. If you ever encounter a compiler with peculiar built-in types, these definitions are the "hook" for making any necessary changes. The complete list of such definitions is

NR Type	*Usual Definition*	*Intent*
`Char`	`char`	8-bit signed integer
`Uchar`	`unsigned char`	8-bit unsigned integer
`Int`	`int`	32-bit signed integer
`Uint`	`unsigned int`	32-bit unsigned integer
`Llong`	`long long int`	64-bit signed integer
`Ullong`	`unsigned long long int`	64-bit unsigned integer
`Doub`	`double`	64-bit floating point
`Ldoub`	`long double`	[reserved for future use]
`Complex`	`complex<double>`	$2 \times$ 64-bit floating complex
`Bool`	`bool`	`true` or `false`

An example of when you might need to change the `typedefs` in `nr3.h` is if your compiler's `int` is not 32 bits, or if it doesn't recognize the type `long long int`.

You might need to substitute vendor-specific types like (in the case of Microsoft) `__int32` and `__int64`.

The second level of type-indirection returns us to the discussion of vectors and matrices. The vector and matrix types that appear in *Numerical Recipes* source code are as follows. Vectors: `VecInt`, `VecUint`, `VecChar`, `VecUchar`, `VecCharp`, `VecLlong`, `VecUllong`, `VecDoub`, `VecDoubp`, `VecComplex`, and `VecBool`. Matrices: `MatInt`, `MatUint`, `MatChar`, `MatUchar`, `MatLlong`, `MatUllong`, `MatDoub`, `MatComplex`, and `MatBool`. These should all be understandable, semantically, as vectors and matrices whose elements are the corresponding user-defined types, above. Those ending in a "p" have elements that are pointers, e.g., `VecCharp` is a vector of pointers to `char`, that is, `char*`. If you are wondering why the above list is not combinatorially complete, it is because we don't happen to use all possible combinations of `Vec`, `Mat`, fundamental type, and pointer in this book. You can add further analogous types as you need them.

Wait, there's more! For every vector and matrix type above, we also define types with the same names plus one of the suffixes "`_I`", "`_O`", and "`_IO`", for example `VecDoub_IO`. We use these suffixed types for specifying argument types in function definitions. The meaning, respectively, is that the argument is "input", "output", or "both input and output".* The `_I` types are automatically defined to be `const`. We discuss this further in §1.5.2 under the topic of `const` correctness.

It may seem capricious for us to define such a long list of types when a much smaller number of templated types would do. The rationale is flexibility: You have a hook into redefining each and every one of the types individually, according to your needs for program efficiency, local coding standards, const-correctness, or whatever. In fact, in `nr3.h`, all these types *are* `typedef`'d to one vector and one matrix class, along the following lines:

```
typedef NRvector<Int> VecInt, VecInt_O, VecInt_IO;
typedef const NRvector<Int> VecInt_I;
...
typedef NRvector<Doub> VecDoub, VecDoub_O, VecDoub_IO;
typedef const NRvector<Doub> VecDoub_I;
...
typedef NRmatrix<Int> MatInt, MatInt_O, MatInt_IO;
typedef const NRmatrix<Int> MatInt_I;
...
typedef NRmatrix<Doub> MatDoub, MatDoub_O, MatDoub_IO;
typedef const NRmatrix<Doub> MatDoub_I;
...
```

So (flexibility, again) you can change the definition of one particular type, like `VecDoub`, or else you can change the implementation of all vectors by changing the definition of `NRvector<>`. Or, you can just leave things the way we have them in `nr3.h`. That ought to work fine in 99.9% of all applications.

1.4.2 Required Methods for Vector and Matrix Classes

The important thing about the vector and matrix classes is not what names they are `typedef`'d to, but what methods are assumed for them (and are provided in the `NRvector` and `NRmatrix` template classes). For vectors, the assumed methods are a

*This is a bit of history, and derives from Fortran 90's very useful `INTENT` attributes.

subset of those in the C++ Standard Library `vector<>` class. If `v` is a vector of type `NRvector<T>`, then we assume the methods:

`v()`	Constructor, zero-length vector.
`v(Int n)`	Constructor, vector of length `n`.
`v(Int n, const T &a)`	Constructor, initialize all elements to the value `a`.
`v(Int n, const T *a)`	Constructor, initialize elements to values in a C-style array, `a[0]`, `a[1]`, ...
`v(const NRvector &rhs)`	Copy constructor.
`v.size()`	Returns number of elements in `v`.
`v.resize(Int newn)`	Resizes `v` to size `newn`. We do not assume that contents are preserved.
`v.assign(Int newn, const T &a)`	Resize `v` to size `newn`, and set all elements to the value `a`.
`v[Int i]`	Element of `v` by subscript, either an l-value and an r-value.
`v = rhs`	Assignment operator. Resizes `v` if necessary and makes it a copy of the vector `rhs`.
`typedef T value_type;`	Makes `T` available externally (useful in templated functions or classes).

As we will discuss later in more detail, you can use any vector class you like with *Numerical Recipes*, as long as it provides the above basic functionality. For example, a brute force way to use the C++ Standard Library `vector<>` class instead of `NRvector` is by the preprocessor directive

```
#define NRvector vector
```

(In fact, there is a compiler switch, `_USESTDVECTOR_`, in the file `nr3.h` that will do just this.)

The methods for matrices are closely analogous. If `vv` is a matrix of type `NRmatrix<T>`, then we assume the methods:

`vv()`	Constructor, zero-length vector.
`vv(Int n, Int m)`	Constructor, $n \times m$ matrix.
`vv(Int n, Int m, const T &a)`	Constructor, initialize all elements to the value `a`.
`vv(Int n, Int m, const T *a)`	Constructor, initialize elements by rows to the values in a C-style array.
`vv(const NRmatrix &rhs)`	Copy constructor.
`vv.nrows()`	Returns number of rows `n`.
`vv.ncols()`	Returns number of columns `m`.
`vv.resize(Int newn, Int newm)`	Resizes `vv` to `newn`$\times$`newm`. We do not assume that contents are preserved.
`vv.assign(Int newn, Int newm, const t &a)`	Resizes `vv` to `newn` $\times$ `newm`, and sets all elements to the value `a`.
`vv[Int i]`	Return a pointer to the first element in row `i` (not often used by itself).
`v[Int i][Int j]`	Element of `vv` by subscript, either an l-value and an r-value.
`vv = rhs`	Assignment operator. Resizes `vv` if necessary and makes it a copy of the matrix `rhs`.
`typedef T value_type;`	Makes `T` available externally.

For more precise specifications, see §1.4.3.

There is one additional property that we assume of the vector and matrix classes, namely that all of an object's elements are stored in sequential order. For a vector, this means that its elements can be addressed by pointer arithmetic relative to the first element. For example, if we have

```
VecDoub a(100);
Doub *b = &a[0];
```

then `a[i]` and `b[i]` reference the same element, both as an l-value and as an r-value. This capability is sometimes important for inner-loop efficiency, and it is also useful for interfacing with legacy code that can handle `Doub*` arrays, but not `VecDoub` vectors. Although the original C++ Standard Library did not guarantee this behavior, all known implementations of it do so, and the behavior is now required by an amendment to the standard [2].

For matrices, we analogously assume that storage is by rows within a single sequential block so that, for example,

```
Int n=97, m=103;
MatDoub a(n,m);
Doub *b = &a[0][0];
```

implies that `a[i][j]` and `b[m*i+j]` are equivalent.

A few of our routines need the capability of taking as an argument either a vector or else one row of a matrix. For simplicity, we usually code this using overloading, as, e.g.,

```
void someroutine(Doub *v, Int m) {          Version for a matrix row.
    ...
}
inline void someroutine(VecDoub &v) {       Version for a vector.
    someroutine(&v[0],v.size());
}
```

For a vector `v`, a call looks like `someroutine(v)`, while for row `i` of a matrix `vv` it is `someroutine(&vv[i][0],vv.ncols())`. While the simpler argument `vv[i]` would in fact work in our implementation of `NRmatrix`, it might not work in some other matrix class that guarantees sequential storage but has the return type for a single subscript different from `T*`.

1.4.3 Implementations in nr3.h

For reference, here is a complete declaration of `NRvector`.

```
template <class T>
class NRvector {
private:
    int nn;                                          Size of array, indices 0..nn-1.
    T *v;                                            Pointer to data array.
public:
    NRvector();                                      Default constructor.
    explicit NRvector(int n);                        Construct vector of size n.
    NRvector(int n, const T &a);                     Initialize to constant value a.
    NRvector(int n, const T *a);                     Initialize to values in C-style array a.
    NRvector(const NRvector &rhs);                   Copy constructor.
    NRvector & operator=(const NRvector &rhs);       Assignment operator.
    typedef T value_type;                            Make T available.
    inline T & operator[](const int i);              Return element number i.
    inline const T & operator[](const int i) const;                  const version.
    inline int size() const;                         Return size of vector.
    void resize(int newn);                           Resize, losing contents.
    void assign(int newn, const T &a);               Resize and assign a to every element.
    ~NRvector();                                     Destructor.
};
```

The implementations are straightforward and can be found in the file nr3.h. The only issues requiring finesse are the consistent treatment of zero-length vectors and the avoidance of unnecessary resize operations.

A complete declaration of NRmatrix is

```
template <class T>
class NRmatrix {
private:
    int nn;                                          Number of rows and columns. Index
    int mm;                                              range is 0..nn-1, 0..mm-1.
    T **v;                                           Storage for data.
public:
    NRmatrix();                                      Default constructor.
    NRmatrix(int n, int m);                          Construct n × m matrix.
    NRmatrix(int n, int m, const T &a);              Initialize to constant value a.
    NRmatrix(int n, int m, const T *a);              Initialize to values in C-style array a.
    NRmatrix(const NRmatrix &rhs);                   Copy constructor.
    NRmatrix & operator=(const NRmatrix &rhs);       Assignment operator.
    typedef T value_type;                            Make T available.
    inline T* operator[](const int i);               Subscripting: pointer to row i.
    inline const T* operator[](const int i) const;                  const version.
    inline int nrows() const;                        Return number of rows.
    inline int ncols() const;                        Return number of columns.
    void resize(int newn, int newm);                 Resize, losing contents.
    void assign(int newn, int newm, const T &a);     Resize and assign a to every element.
    ~NRmatrix();                                     Destructor.
};
```

A couple of implementation details in NRmatrix are worth commenting on. The private variable **v points not to the data but rather to an array of pointers to the data rows. Memory allocation of this array is separate from the allocation of space for the actual data. The data space is allocated as a single block, not separately for each row. For matrices of zero size, we have to account for the separate possibilities that there are zero rows, or that there are a finite number of rows, but each with zero columns. So, for example, one of the constructors looks like this:

```
template <class T>
NRmatrix<T>::NRmatrix(int n, int m) : nn(n), mm(m), v(n>0 ? new T*[n] : NULL)
{
    int i,nel=m*n;
    if (v) v[0] = nel>0 ? new T[nel] : NULL;
    for (i=1;i<n;i++) v[i] = v[i-1] + m;
}
```

Finally, it matters *a lot* whether your compiler honors the inline directives in NRvector and NRmatrix above. If it doesn't, then you may be doing full function calls, saving and restoring context within the processor, every time you address a vector or matrix element. This is tantamount to making C++ useless for most numerical computing! Luckily, as we write, the most commonly used compilers are all "honorable" in this respect.

CITED REFERENCES AND FURTHER READING:

Josuttis, N.M. 1999, *The C++ Standard Library: A Tutorial and Reference* (Boston: Addison-Wesley).[1]

International Standardization Organization 2003, *Technical Corrigendum ISO 14882:2003*.[2]

1.5 Some Further Conventions and Capabilities

We collect in this section some further explanation of C++ language capabilities and how we use them in this book.

1.5.1 Error and Exception Handling

We already mentioned that we code error conditions with simple `throw` statements, like this

```
throw("error foo in routine bah");
```

If you are programming in an environment that has a defined set of error classes, and you want to use them, then you'll need to change these lines in the routines that you use. Alternatively, without any additional machinery, you can choose between a couple of different, useful behaviors just by making small changes in `nr3.h`.

By default, `nr3.h` redefines `throw()` by a preprocessor macro,

```
#define throw(message) \
    {printf("ERROR: %s\n in file %s at line %d\n", \
     message,__FILE__,__LINE__); \
     exit(1);}
```

This uses standard ANSI C features, also present in C++, to print the source code file name and line number at which the error occurs. It is inelegant, but perfectly functional.

Somewhat more functional, and definitely more elegant, is to set `nr3.h`'s compiler switch `_USENRERRORCLASS_`, which instead substitutes the following code:

```
struct NRerror {
    char *message;
    char *file;
    int line;
    NRerror(char *m, char *f, int l) : message(m), file(f), line(l) {}
};

void NRcatch(NRerror err) {
    printf("ERROR: %s\n     in file %s at line %d\n",
        err.message, err.file, err.line);
    exit(1);
}
```

```
#define throw(message) throw(NRerror(message,__FILE__,__LINE__));
```

Now you have a (rudimentary) error class, `NRerror`, available. You use it by putting a `try...catch` control structure at any desired point (or points) in your code, for example (§2.9),

```
...
try {
    Cholesky achol(a);
}
catch (NRerror err) {
    NRcatch(err);
}
```

Executed if `Cholesky` throws an exception.

As shown, the use of the `NRcatch` function above simply mimics the behavior of the previous macro in the global context. But you don't have to use `NRcatch` at all: You

can substitute any code that you want for the body of the `catch` statement. If you want to distinguish between different kinds of exceptions that may be thrown, you can use the information returned in `err`. We'll let you figure this out yourself. And of course you are welcome to add more complicated error classes to your own copy of `nr3.h`.

1.5.2 Const Correctness

Few topics in discussions about C++ evoke more heat than questions about the keyword `const`. We are firm believers in using `const` wherever possible, to achieve what is called "`const` correctness." Many coding errors are automatically trapped by the compiler if you have qualified identifiers that should not change with `const` when they are declared. Also, using `const` makes your code much more readable: When you see `const` in front of an argument to a function, you know immediately that the function will not modify the object. Conversely, if `const` is absent, you should be able to count on the object being changed somewhere.

We are such strong `const` believers that we insert `const` even where it is theoretically redundant: If an argument is passed *by value* to a function, then the function makes a copy of it. Even if this copy is modified by the function, the original value is unchanged after the function exits. While this allows you to change, with impunity, the values of arguments that have been passed by value, this usage is error-prone and hard to read. If your intention in passing something by value is that it is an input variable only, then make it clear. So we declare a function $f(x)$ as, for example,

```
Doub f(const Doub x);
```

If in the function you want to use a local variable that is initialized to `x` but then gets changed, define a new quantity — don't use `x`. If you put `const` in the declaration, the compiler will not let you get this wrong.

Using `const` in your function arguments makes your function more general: Calling a function that expects a `const` argument with a non-`const` variable involves a "trivial" conversion. But trying to pass a `const` quantity to a non-`const` argument is an error.

A final reason for using `const` is that it allows certain user-defined conversions to be made. As discussed in [1], this can be useful if you want to use *Numerical Recipes* routines with another matrix/vector class library.

We now need to elaborate on what exactly `const` does for a nonsimple type such as a class that is an argument of a function. Basically, it guarantees that the object is not modified by the function. In other words, the object's data members are unchanged. But if a data member is a *pointer* to some data, and the data itself is not a member variable, then *the data can be changed* even though the pointer cannot be.

Let's look at the implications of this for a function `f` that takes an `NRvector<Doub>` argument `a`. To avoid unnecessary copying, we always pass vectors and matrices by reference. Consider the difference between declaring the argument of a function with and without `const`:

```
void f(NRvector<Doub> &a)        versus        void f(const NRvector<Doub> &a)
```

The `const` version promises that `f` does not modify the data members of `a`. But a statement like

```
a[i] = 4.;
```

inside the function definition is in principle perfectly OK — you are modifying the data pointed to, not the pointer itself.

"Isn't there some way to protect the data?" you may ask. Yes, there is: You can declare the *return type* of the subscript operator, `operator[]`, to be `const`. This is why there are two versions of `operator[]` in the `NRvector` class,

```
T & operator[](const int i);
const T & operator[](const int i) const;
```

The first form returns a reference to a modifiable vector element, while the second returns a nonmodifiable vector element (because the return type has a `const` in front).

But how does the compiler know which version to invoke when you just write `a[i]`? That is specified by the *trailing* word `const` in the second version. It refers not to the returned element, nor to the argument `i`, but to the object whose `operator[]` is being invoked, in our example the vector `a`. Taken together, the two versions say this to the compiler: "If the vector `a` is `const`, then transfer that const'ness to the returned element `a[i]`. If it isn't, then don't."

The remaining question is thus how the compiler determines whether `a` is `const`. In our example, where `a` is a function argument, it is trivial: The argument is either declared as `const` or else it isn't. In other contexts, `a` might be `const` because you originally declared it as such (and initialized it via constructor arguments), or because it is a `const` reference data member in some other object, or for some other, more arcane, reason.

As you can see, getting `const` to protect the data is a little complicated. Judging from the large number of matrix/vector libraries that follow this scheme, many people feel that the payoff is worthwhile. We urge you *always* to declare as `const` those objects and variables that are not intended to be modified. You do this both at the time an object is actually created and in the arguments of function declarations and definitions. You won't regret making a habit of `const` correctness.

In §1.4 we defined vector and matrix type names with trailing `_I` labels, for example, `VecDoub_I` and `MatInt_I`. The `_I`, which stands for "input to a function," means that the type is declared as `const`. (This is already done in the `typedef` statement; you don't have to repeat it.) The corresponding labels `_O` and `_IO` are to remind you when arguments are not just non-`const`, but will actually be modified by the function in whose argument list they appear.

Having rightly put all this emphasis on `const` correctness, duty compels us also to recognize the existence of an alternative philosophy, which is to stick with the more rudimentary view "`const` protects the container, not the contents." In this case you would want only *one* form of `operator[]`, namely

```
T & operator[](const int i) const;
```

It would be invoked whether your vector was passed by `const` reference or not. In both cases element `i` is returned as potentially modifiable. While we are opposed to this philosophically, it turns out that it does make possible a tricky kind of automatic type conversion that allows you to use your favorite vector and matrix classes instead of `NRvector` and `NRmatrix`, even if your classes use a syntax completely different from what we have assumed. For information on this very specialized application, see [1].

1.5.3 Abstract Base Class (ABC), or Template?

There is sometimes more than one good way to achieve some end in C++. Heck, let's be honest: There is *always* more than one way. Sometimes the differences amount to small tweaks, but at other times they embody very different views about the language. When we make one such choice, and you prefer another, you are going to be quite annoyed with us. Our defense against this is to avoid foolish consistencies,* and to illustrate as many viewpoints as possible.

A good example is the question of when to use an abstract base class (ABC) versus a template, when their capabilities overlap. Suppose we have a function `func` that can do its (useful) thing on, or using, several different types of objects, call them `ObjA`, `ObjB`, and `ObjC`. Moreover, `func` doesn't need to know much about the object it interacts with, only that it has some method `tellme`.

We could implement this setup as an abstract base class:

```
struct ObjABC {                          Abstract Base Class for objects with tellme.
    virtual void tellme() = 0;
};

struct ObjA : ObjABC {                   Derived class.
    ...
    void tellme() {...}
};
struct ObjB : ObjABC {                   Derived class.
    ...
    void tellme() {...}
};
struct ObjC : ObjABC {                   Derived class.
    ...
    void tellme() {...}
};

void func(ObjABC &x) {
    ...
    x.tellme();                          References the appropriate tellme.
}
```

On the other hand, using a template, we can write code for `func` without ever seeing (or even knowing the names of) the objects for which it is intended:

```
template<class T>
void func(T &x) {
    ...
    x.tellme();
}
```

That certainly seems easier! Is it better?

Maybe. A disadvantage of templates is that the template must be available to the compiler every time it encounters a call to `func`. This is because it actually compiles a different version of `func` for every different type of argument `T` that it encounters. Unless your code is so large that it cannot easily be compiled as a single unit, however, this is not much of a disadvantage. On the other side, favoring templates, is the fact that virtual functions incur a small run-time penalty when they are called. But this is rarely significant.

The deciding factors in this example relate to software engineering, not performance, and are hidden in the lines with ellipses (. . .). We haven't really told

*"A foolish consistency is the hobgoblin of little minds." —Emerson

you how closely related ObjA, ObjB, and ObjC are. If they are close, then the ABC approach offers possibilities for putting more than just tellme into the base class. Putting things into the base class, whether data or pure virtual methods, lets the compiler enforce consistency across the derived classes. If you later write another derived object ObjD, its consistency will also be enforced. For example, the compiler will require you to implement a method in every derived class corresponding to every pure virtual method in the base class.

By contrast, in the template approach, the only enforced consistency will be that the method tellme exists, and this will only be enforced at the point in the code where func is actually called with an ObjD argument (if such a point exists), not at the point where ObjD is defined. Consistency checking in the template approach is thus somewhat more haphazard.

Laid-back programmers will opt for templates. Up-tight programmers will opt for ABCs. We opt for... both, on different occasions. There can also be other reasons, having to do with subtle features of class derivation or of templates, for choosing one approach over the other. We will point these out as we encounter them in later chapters. For example, in Chapter 17 we define an abstract base class called StepperBase for the various "stepper" routines for solving ODEs. The derived classes implement particular stepping algorithms, and they are templated so they can accept function or functor arguments (see §1.3.3).

1.5.4 NaN and Floating Point Exceptions

We mentioned in §1.1.1 that the IEEE floating-point standard includes a representation for NaN, meaning "not a number." NaN is distinct from positive and negative infinity, as well as from every representable number. It can be both a blessing and a curse.

The blessing is that it can be useful to have a value that can be used with meanings like "don't process me" or "missing data" or "not yet initialized." To use NaN in this fashion, you need to be able to *set* variables to it, and you need to be able to *test* for its having been set.

Setting is easy. The "approved" method is to use numeric_limits. In nr3.h the line

```
static const Doub NaN = numeric_limits<Doub>::quiet_NaN();
```

defines a global value NaN, so that you can write things like

```
x = NaN;
```

at will. If you ever encounter a compiler that doesn't do this right (it's a pretty obscure corner of the standard library!), then try either

```
Uint proto_nan[2]=0xffffffff, 0x7fffffff;
double NaN = *( double* )proto_nan;
```

(which assumes little-endian behavior; cf. §1.1.1) or the self-explanatory

```
Doub NaN = sqrt(-1.);
```

which may, however, throw an immediate exception (see below) and thus not work for this purpose. But, one way or another, you can generally figure out how to get a NaN constant into your environment.

Testing also requires a bit of (one-time) experimentation: According to the IEEE standard, NaN is guaranteed to be the only value that is not equal to itself!

So, the "approved" method of testing whether Doub value x has been set to NaN is

```
if (x != x) {...}          It's a NaN!
```

(or test for equality to determine that it's not a NaN). Unfortunately, at time of writing, some otherwise perfectly good compilers don't do this right. Instead, they provide a macro `isnan()` that returns `true` if the argument is NaN, otherwise `false`. (Check carefully whether the required `#include` is `math.h` or `float.h` — it varies.)

What, then, is the *curse* of NaN? It is that some compilers, notably Microsoft, have poorly thought-out default behaviors in distinguishing *quiet NaNs* from *signalling NaNs*. Both kinds of NaNs are defined in the IEEE floating-point standard. Quiet NaNs are supposed to be for uses like those above: You can set them, test them, and propagate them by assignment, or even through other floating operations. In such uses they are not supposed to signal an exception that causes your program to abort. Signalling NaNs, on the other hand, are, as the name implies, supposed to signal exceptions. Signalling NaNs should be generated by invalid operations, such as the square root or logarithm of a negative number, or `pow(0.,0.)`.

If all NaNs are treated as signalling exceptions, then you can't make use of them as we have suggested above. That's annoying, but OK. On the other hand, if all NaNs are treated as quiet (the Microsoft default at time of writing), then you will run long calculations only to find that all the results are NaN — and you have no way of locating the invalid operation that triggered the propagating cascade of (quiet) NaNs. That's *not* OK. It makes debugging a nightmare. (You can get the same disease if other floating-point exceptions propagate, for example overflow or division-by-zero.)

Tricks for specific compilers are not within our normal scope. But this one is so essential that we make it an "exception": If you are living on planet Microsoft, then the lines of code,

```
int cw = _controlfp(0,0);
cw &=~(EM_INVALID | EM_OVERFLOW | EM_ZERODIVIDE );
_controlfp(cw,MCW_EM);
```

at the beginning of your program will turn NaNs from invalid operations, overflows, and divides-by-zero into signalling NaNs, and leave all the other NaNs quiet. There is a compiler switch, `_TURNONFPES_` in `nr3.h` that will do this for you automatically. (Further options are `EM_UNDERFLOW`, `EM_INEXACT`, and `EM_DENORMAL`, but we think these are best left quiet.)

1.5.5 Miscellany

- Bounds checking in vectors and matrices, that is, verifying that subscripts are in range, is expensive. It can easily double or triple the access time to subscripted elements. In their default configuration, the `NRvector` and `NRmatrix` classes never do bounds checking. However, `nr3.h` has a compiler switch, `_CHECKBOUNDS_`, that turns bounds checking on. This is implemented by preprocessor directives for conditional compilation so there is no performance penalty when you leave it turned off. This is ugly, but effective.

The `vector<>` class in the C++ Standard Library takes a different tack. If you access a vector element by the syntax `v[i]`, there is no bounds checking. If you instead use the `at()` method, as `v.at(i)`, then bounds checking is performed. The obvious weakness of this approach is that you can't easily change a lengthy program from one method to the other, as you might want to

do when debugging.9

- The importance to performance of avoiding unnecessary copying of large objects, such as vectors and matrices, cannot be overemphasized. As already mentioned, they should always be passed by reference in function arguments. But you also need to be careful about, or avoid completely, the use of functions whose return type is a large object. This is true even if the return type is a reference (which is a tricky business anyway). Our experience is that compilers often create temporary objects, using the copy constructor, when the need to do so is obscure or nonexistent. That is why we so frequently write `void` functions that have an argument of type (e.g.) `MatDoub_O` for returning the "answer." (When we do use vector or matrix return types, our excuse is either that the code is pedagogical, or that the overhead is negligible compared to some big calculation that has just been done.)1

  You can check up on your compiler by instrumenting the vector and matrix classes: Add a static integer variable to the class definition, increment it within the copy constructor and assignment operator, and look at its value before and after operations that (you think) should not require any copies. You might be surprised.3

- There are only two routines in *Numerical Recipes* that use three-dimensional arrays, `rlft3` in §12.6, and `solvde` in §18.3. The file `nr3.h` includes a rudimentary class for three-dimensional arrays, mainly to service these two routines. In many applications, a better way to proceed is to declare a vector of matrices, for example,2

```
vector<MatDoub> threedee(17);11
for (Int i=0;i<17;i++) threedee[i].resize(19,21);10
```

  which creates, in effect, a three-dimensional array of size $17 \times 19 \times 21$. You can address individual components as `threedee[i][j][k]`.7

- "Why no `namespace`?" Industrial-strength programmers will notice that, unlike the second edition, this third edition of *Numerical Recipes* does not shield its function and class names by a `NR::` namespace. Therefore, if you are so bold as to `#include` every single file in the book, you are dumping on the order of 500 names into the global namespace, definitely a bad idea!4

  The explanation, quite simply, is that the vast majority of our users are not industrial-strength programmers, and most found the `NR::` namespace annoying and confusing. As we emphasized, strongly, in §1.0.1, NR is not a program library. If you want to create your own personal namespace for NR, please go ahead.6

- In the distant past, *Numerical Recipes* included provisions for unit- or one-based, instead of zero-based, array indices. The last such version was published in 1992. Zero-based arrays have become so universally accepted that we no longer support any other option.5

CITED REFERENCES AND FURTHER READING:1

Numerical Recipes Software 2007, "Using Other Vector and Matrix Libraries," *Numerical Recipes Webnote No. 1*, at `http://www.nr.com/webnotes?1` [1]8