

# Chapter 1. Preliminaries

## 1.0 Introduction

This book, like its sibling versions in other computer languages, 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.

We presume that you are able to read computer programs in C++, that being the language of this version of *Numerical Recipes*. The books *Numerical Recipes in Fortran 77*, *Numerical Recipes in Fortran 90*, and *Numerical Recipes in C* are separately available, if you prefer to program in one of those languages. Earlier editions of *Numerical Recipes in Pascal* and *Numerical Recipes Routines and Examples in BASIC* are also available; while not containing the additional material of the Second Edition versions, these versions are perfectly serviceable if Pascal or BASIC is your language of choice.

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

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::flmoon(const int n, const int nph, int &jd, DP &frac)
Our programs 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 nth such phase since January, 1900. Greenwich Mean Time is assumed.
{
    const DP RAD=3.141592653589793238/180.0;
    int i;
```

```

DP am,as,c,t,t2,xtra;

c=n+nph/4.0;
t=c/1236.85;
t2=t*t;
as=359.2242+29.105356*c;
am=306.0253+385.816918*c+0.010730*t2;
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 nrerror("nph is unknown in flmoon");
i=int(xtra >= 0.0 ? floor(xtra) : ceil(xtra-1.0));
jd += i;
frac=xtra-i;
}

```

This is how we comment an individual line.

You aren't really intended to understand this algorithm, but it does work!

This is how we will indicate error conditions.

Note our convention of handling all errors and exceptional cases with a statement like `nrerror("some error message");`. The function `nrerror()` is part of a small file of utility programs, `nrutil.h`, listed in Appendix B at the back of the book. This Appendix includes a number of other utilities that we will describe later in this chapter. Function `nrerror()` prints the indicated error message to your `stderr` device (usually your terminal screen), and then invokes the function `exit()`, which terminates execution. You can modify `nrerror()` so that it does anything else that will halt execution. For example, you can have it pause for input from the keyboard, and then manually interrupt execution. In some applications, you will want to modify `nrerror()` to do more sophisticated error handling, for example to transfer control somewhere else by throwing a C++ exception.

We will have more to say about the C++ programming language, its conventions and style, in §1.1 and §1.2.

### **Quick Start: Using the C++ Numerical Recipes Routines**

This section is for people who want to jump right in. We'll 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!)

First, locate the important files `nrtypes.h`, `nrutil.h`, and `nr.h`, as listed in Appendices A and B. These contain the definitions of the various types used by our routines, the vector and matrix classes we use, various utility functions, and the function declarations for all the Recipe functions. (Actually, `nrtypes.h` includes by default the file `nrtypes_nr.h`, and `nrutil.h` includes by default the file `nrutil_nr.h`. This setup is to allow you to change the defaults easily, as will be discussed in §1.3.)

Second, create this main program file:

```

#include <iostream>
#include <iomanip>
#include "nr.h"
using namespace std;

int main(void)
{

```

```
const int NTOT=20;
int i,jd,nph=2;
DP frac,ave,vrnce;
Vec_DP data(NTOT);

for (i=0;i<NTOT;i++) {
    NR::flmoon(i,nph,jd,frac);
    data[i]=jd;
}
NR::avevar(data,ave,vrnce);
cout << "Average = " << setw(12) << ave;
cout << " Variance = " << setw(13) << vrnce << endl;
return 0;
}
```

Here is a brief explanation of some elements of the above program:

You must always have the `#include nr.h` statement, which includes the function declarations. It also includes the files `nrtypes.h` and `nrutil.h` for you. The declaration `Vec_DP data(NTOT)` creates a double precision vector with `NTOT` elements. The types `DP` (double precision, i.e., `double`) and `Vec_DP` are defined in `nrtypes.h`, while the vector class is defined in `nrutil.h`. The scope resolution prefix `NR::` makes accessible the Recipe functions `flmoon` and `avevar`, which are in the `NR` namespace defined in `nr.h`. We call `flmoon`, looping over 20 full moons, and store the Julian days it returns in `data`. Then we call the `avevar` routine, and print the answers.

Third, compile the main program file, and also the files `flmoon.cpp` and `avevar.cpp`. Link the resulting object files.

Fourth, run the resulting executable file. Typical output is:

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

The files `nrtypes.h`, `nrutil.h`, and `nr.h` and the concepts behind them will be discussed in detail in §1.2 and 1.3.

## ***Computational Environment and Program Validation***

Our goal is that the programs in this book be as portable as possible, across different platforms (models of computer), across different operating systems, and across different C++ compilers. C++ was designed with this type of portability in mind. Nevertheless, we have found that there is no substitute for actually checking all programs on a variety of compilers, in the process uncovering differences in library structure or contents, and even occasional differences in allowed syntax. As surrogates for the large number of hardware and software configurations, we have tested all the programs in this book on the combinations of machines, operating systems, and compilers shown on the accompanying table. More generally, the programs should run without modification on any compiler that implements the ANSI/ISO C++ standard, as described for example in Stroustrup's excellent book [1].

In validating the programs, we have taken the program source code directly from the machine-readable form of the book's manuscript, to decrease the chance of propagating typographical errors. "Driver" or demonstration programs that we used as part of our validations are available separately as the *Numerical Recipes Example Book (C++)*, as well as in machine-readable form. If you plan to use more than a few of the programs in this book, then you may find it useful to obtain

Tested Machines and Compilers	
O/S and Hardware	Compiler Version
Microsoft Windows / Intel	Microsoft Visual C++ 6.0
Microsoft Windows / Intel	Borland C++ 5.02
Linux / Intel	GNU C++ ("g++") 2.95.2
AIX 4.3 / IBM RS/6000	KAI C++ ("KCC") 4.0d
SunOS 5.7 (Solaris 7) / Sun Ultra 5	Sun Workshop Compiler C++ ("CC") 5.0
SunOS 5.7 (Solaris 7) / Sun Ultra 5	GNU C++ ("g++") 2.95.2

the machine-readable software distribution, which includes both the Recipes and the demonstration programs.

Of course we would be foolish to claim that there are no bugs in our programs, and we do not make such a claim. We have been very careful, and have benefitted from the experience of the many readers who have written to us. If you find a new bug, please document it and tell us! You can find contact information at <http://www.nr.com>.

### **Compatibility with the First Edition**

If you are accustomed to the *Numerical Recipes* routines of the First Edition, rest assured: almost all of them are still here, with the same names and functionalities, often with major improvements in the code itself. In addition, we hope that you will soon become equally familiar with the added capabilities of the more than 100 routines that are new to this edition.

We have retired a small number of First Edition routines, those that we believe to be clearly dominated by better methods implemented in this edition. A table, following, lists the retired routines and suggests replacements.

Previous Routines Omitted from This Edition		
Name(s)	Replacement(s)	Comment
adi	mglin or mgfas	better method
cosft	cosft1 or cosft2	choice of boundary conditions
cel, el2	rf, rd, rj, rc	better algorithms
des, desks	ran4 now uses psdes	was too slow
iindexx	indexx	indexx overloaded for double and int
mdian1, mdian2	select, selip	more general
qcksrt	sort	name change (sort is now hpsort)
rkqc	rkqs	better method
smooft	use convlv with coefficients from savgol	
sparse	linbcg	more general

- Wirth, N. 1983, *Programming in Modula-2*, 3rd ed. (New York: Springer-Verlag). [4]
- Stroustrup, B. 1997, *The C++ Programming Language*, 3rd ed. (Reading, MA: Addison-Wesley). [5]
- Meeus, J. 1982, *Astronomical Formulae for Calculators*, 2nd ed., revised and enlarged (Richmond, VA: Willmann-Bell). [6]
- Hatcher, D.A. 1984, *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. [7]

## 1.2 Some C++ Conventions for Scientific Computing

The C++ language derives from C, which in turn was originally devised for systems programming work, not for scientific computing. Relative to other high-level programming languages, C puts the programmer “very close to the machine” in several respects. It is operator-rich, giving direct access to most capabilities of a machine-language instruction set. It has a large variety of intrinsic data types (short and long, signed and unsigned integers; floating and double-precision reals; pointer types; etc.), and a concise syntax for effecting conversions and indirections. It defines an arithmetic on pointers (addresses) that relates gracefully to array addressing and is highly compatible with the index register structure of many computers. All these features are present in C++ too.

Portability is another strong point of the C language, and so also of C++. C is the underlying language of the UNIX and Linux operating systems; both the language and these operating systems have by now been implemented on literally hundreds of different computer types. The language’s universality, portability, and flexibility have attracted increasing numbers of scientists and engineers to it. It is commonly used for the real-time control of experimental hardware, often in spite of the fact that the standard UNIX or Linux kernel is less than ideal as an operating system for this purpose.

The use of C or C++ for higher level scientific calculations such as data analysis, modeling, and floating-point numerical work has generally been slower in developing. In part this is because of the entrenched position of Fortran as the mother-tongue of virtually all scientists and engineers born before 1960, and many born after. In part, also, the slowness of C’s penetration into scientific computing has been due to deficiencies in the language that computer scientists have been (we think, stubbornly) slow to recognize. Examples are the lack of a good way to raise numbers to small integer powers, and the difficulty in getting compilers to emit optimized code when high-level constructs are used.

The C version of this book attempted to lay out by example a set of sensible, practical conventions for scientific C programming. In this version we will try to do the same for C++.

The need for programming conventions in C++ is very great. Far from the problem of overcoming constraints imposed by the language (our repeated experience with Pascal), the problem in C++ is to choose the best and most natural techniques from multiple opportunities — and then to use those techniques

completely consistently from program to program. In the rest of this section, we set out some of the issues, and describe the adopted conventions that are used in all of the routines in this book. The best general reference for detailed information on the C++ language is Stroustrup's book [1].

## **Double Precision**

When the C version of this book first came out, the default precision used by most scientists for most calculations was single precision. The reason was that double precision imposed a significant overhead both in execution speed and in memory requirements. (The fact that C automatically converts `float` variables to `double` in many situations was another insult to scientific programmers!)

Nowadays, the execution speed overhead has essentially disappeared. There are even machines where single precision is slower than double! And memory conservation is often not the concern it used to be. Accordingly, the default precision in this C++ edition is double precision. To make it easy to change to single precision (or quadruple precision!) if you want to, we have not hard coded the type `double` in the routines. Instead we have used the name `DP`, along with the `typedef` definition

```
typedef double DP;
```

This `typedef` occurs twice, once in `nrtypes.h` and once in `nrutil.h`. To change the default precision to single, just change the definition in both places to

```
typedef float DP;
```

You will also have to change the values of certain “accuracy parameters” in some routines. This is further described in Appendix C. For some of the Recipes roundoff error is a particular concern. In those cases we will explicitly warn you always to use double precision.

## **Function Declarations, Header Files, and Namespaces**

In C++ a function cannot be called unless it has previously been *declared*. A function declaration gives the return type of the function, and the number and types of its arguments. (Function declarations are also called function prototypes or function headers.) For example,

```
int g(int x, int y, double z);           Function declaration.
```

A function must also be *defined* somewhere (and defined once only). A function definition consists of a function declaration plus the body of the function, the code that actually does the work. For example,

```
int g(int x, int y, double z)           Function definition.
{
    return x+int(z)/y;
}
```

If all your code is in one file, you can often arrange for each function definition to precede the functions that call it. Since the definition includes the declaration, the compiler can then check that a given function call invokes the function with the correct argument types. However, this setup is feasible only for small programs. In general, a C++ program consists of multiple source files that are separately compiled, and the compiler cannot check the consistency of each function call without some additional assistance. A simple and safe way to proceed is as follows [1]:

- Every external function should have a single declaration in a header (.h) file.
- The source file with the definition (body) of the function should also include the header file so that the compiler can check that the declaration and the definition match.
- Every source file that calls the function should include the same header file.

For the routines in this book, the header file containing all the declarations is `nr.h`, listed in Appendix A. You should put the statement `#include "nr.h"` at the top of every source file that invokes *Numerical Recipes* routines.

A *namespace* is a means of hiding the implementation of a program module, so that only its interface is accessible from the user program. This provides you with control over which variables and functions are “visible” in other parts of the program, and offers a way to group objects and functions of related purpose into separate modules.

All of the *Numerical Recipes* function declarations are in the NR namespace. Indeed, the file `nr.h` has the following structure:

```
namespace NR {
    void addint(Mat_0_DP &uf, Mat_I_DP &uc, Mat_0_DP &res);
    ...
    void zroots(Vec_I_CPLX_DP &a, Vec_0_CPLX_DP &roots, const bool &polish);
}
```

(The types `Mat_0_DP`, etc. will be explained below.) The *Numerical Recipes* function definitions have the following format:

```
#include "nr.h"

void NR::addint(Mat_0_DP &uf, Mat_I_DP &uc, Mat_0_DP &res)
    ...
```

Note that the code includes `nr.h` (since that is where the namespace NR is defined). Also, the function name must be prefixed with `NR::` since NR is the scope in which the function is being defined. Note that if the function happens to call another *Numerical Recipes* function, no special declaration is required. Since all the Recipes are in namespace NR, they are all within each other’s scope.

When you write a program that calls one of the Recipe functions, your program must include `nr.h`. You then have three ways of invoking a particular Recipe:

- Call the function with explicit scope resolution:

```
NR::addint(uf, uc, res);
```

- Precede the function call with a using declaration:

```
using NR::addint;  
...  
addint(uf, uc, res);
```

- Include a using directive in the program:

```
using namespace NR;  
...  
addint(uf, uc, res);
```

This directive makes *all* the functions in NR visible, something purists would regard as sloppy practice.

Some further details about the file `nr.h` are given in Appendix A.

We mention here another feature of namespaces that we make extensive use of in the Recipes. Often one wants to define an ancillary function that is used by another function. This adjunct function does some task that is of no general interest, so we want its definition to be local to the file in which the principal function is defined. In C you achieve this by defining the function as a `static` function. In C++ it is better to reserve `static` to declare objects inside functions and classes whose storage must be nonvolatile. Local functions hidden from the rest of your program can be put inside an *unnamed namespace*. Only functions within the same file can access objects inside an unnamed namespace (see, e.g., `golden.cpp` in §10.1).

## Const Correctness

Few topics in discussions about C++ evoke more heat than questions about the keyword `const`. Here is our position: 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 routine, you know immediately that the routine 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 many people think it is 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, we believe 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,

```
double f(const double 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.

Some people think that using `const` on arguments makes your functions less general. Quite the opposite! 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.

The final reason for using `const` is that it allows certain user-defined conversions to be made. As we will see in §1.3, this is the key to writing our functions so that you can use them transparently with any matrix/vector class library.

## Vectors and Matrices

Vectors and matrices are the fundamental building blocks of numerical programming. One-dimensional C-style arrays, declared for example as `double b[4]`, are part of C++ and may occasionally be used in numerical work. However, fixed size two-dimensional arrays with declarations like `double a[5][9]` are almost never desirable. Scientific programming requires some mechanism for implementing *variable dimension arrays*, which are passed to a function along with real-time information about their two-dimensional size.

There is no technical reason that a C or C++ compiler could not allow a syntax like

```
void someroutine(a,m,n)
double a[m][n];          /* ILLEGAL DECLARATION */
```

and emit code to evaluate the variable dimensions `m` and `n` (or any variable-dimension expression) each time `someroutine()` is entered. Alas! the above fragment is forbidden by the C++ language definition.

The natural way to deal with vectors and matrices in C++ is as classes. A matrix object can contain not only the values of the matrix elements, but also information on the matrix size. Moreover, the class can provide various overloaded operators and functions to facilitate high-level programming. The C++ standard already provides such a class for vectors (`valarray`), but not for matrices. Many people have written their own class libraries for one- and two-dimensional arrays. Most of these libraries are excellent in providing a suite of high-level constructs, but are terribly inefficient in execution.

When we began preparing this C++ version of *Numerical Recipes*, our original intention was to “do it right:” we would construct a class library with all the necessary high-level constructs that would also be efficient. This turns out to be a highly nontrivial task, especially since many compilers do not yet implement all the features of the C++ standard that are necessary to make such code efficient. Furthermore, we soon realized that this was exactly the wrong way to proceed. Why should you, the reader, adopt *our* class library when you have probably already invested a large effort in developing or using a different one? And what if you want to use our routines in a code written with another class library?

So, instead, we have gone to completely the opposite extreme. The Recipe functions in this book are written with vector and matrix classes called `NRVec` and `NRMat`. These classes are defined in `nrutil.h`, and provide a *minimal*

implementation of vector and matrix operations. You can use our routines in either of two ways:

- Use our classes to handle vectors and matrices in all your programs. When you include `nr.h` to make the Recipes available, it automatically includes `nrutil.h` for you, so our vector and matrix classes will be accessible. The only disadvantage of this is that our classes do not provide many high-level constructs.
- Use any vector and matrix classes you like. In §1.3, we will show you how to set up a modified `nrutil.h` so that you will be able call our routines transparently, even though ours are declared with the Numerical Recipes classes `NRVec` and `NRMat`.

The machinery behind the `NRVec` and `NRMat` classes, and the way they can be used with other class libraries, is quite complicated. Most readers will not want to delve into this material. Accordingly, we defer its discussion to §1.3. All you really need to know to be able to understand how vectors and matrices are used in the Recipes is in the remainder of this section.

## **Vectors and Matrices are Zero-Based**

Arrays in C and C++ are natively “zero-based” or “zero-offset.” An array declared by the statement `double a[3];` has the valid references `a[0]`, `a[1]`, and `a[2]`, but *not* `a[3]`. However, many mathematical algorithms naturally like to go from 1 to  $n$ , not from 0 to  $n - 1$ . In the C version of this book we showed how to use the power of the C language to declare *unit-offset* arrays efficiently and elegantly, and how to use them alongside zero-offset arrays. However, one message that came through loud and clear from our readers’ letters was that they didn’t like unit-offset arrays. Accordingly, *all the routines in this book use zero-based arrays exclusively*. Even where a mathematical algorithm would be more naturally expressed with indices running from 1 to  $n$ , we have presented it in the text with indices from 0 to  $n - 1$ . Thus the indices in the code arrays are the same as those in the text.

## **Vector and Matrix Type Names in the Recipes: `nrtypes.h`**

In §1.3 we describe the nitty-gritty of how our vector and matrix classes are implemented, and how you can alternatively use any other matrix/vector library instead. Many readers will find this tough going. Fortunately, it is possible to insulate you almost entirely from the details of the matrix/vector library. In fact, if you peruse the Recipe functions, you won’t see the class names `NRVec` or `NRMat` appearing anywhere. Instead, you’ll see names like `Vec_I_DP` and `Mat_O_INT` declaring vectors and matrices. These are the identifiers you should actually use, and it is the file `nrtypes.h` that encapsulates all these definitions in a set of `typedef` definitions.

The name we use for a typical type defined in `nrtypes.h` consists of three parts:

- `Vec` or `Mat` for vector or matrix.
- `I`, `O`, or `IO` for in, out, or in-out. These symbols happen to be based on the corresponding Fortran 90 names “intent in”, “intent out”, and “intent inout,” but the concepts are universal. They describe whether the array being passed to the function supplies values to the function but does

not return values (intent in), returns values from the function but does not supply any (intent out), or does both (intent inout).

- the scalar type that is the intended default. For example, INT for int, DP (double precision) for double, CPLX\_SP (complex single precision) for `complex<float>`, and so on. Here is a complete list of all the types we so denote:

BOOL	bool
CHR	char
UCHR	unsigned char
INT	int
UINT	unsigned int
LNG	long
ULNG	unsigned long
SP	float
DP	double
CPLX_SP	<code>complex&lt;float&gt;</code>
CPLX_DP	<code>complex&lt;double&gt;</code>
ULNG_p	unsigned long *
DP_p	double *
FSTREAM_p	fstream *

A sample entry in `nrtypes.h` for double precision vectors is

```
typedef const NRVec<DP> Vec_I_DP;
typedef NRVec<DP> Vec_DP, Vec_O_DP, Vec_IO_DP;
```

The first line says that intent in vectors will be `const` double precision. The second line says that intent out and intent inout vectors will not be `const`. The identifier `Vec_DP` is for vectors declared locally within a Recipe function, where the terms “in” and “out” are not meaningful.

With these defined types, you can write programs completely without reference to the implementation of the underlying vector and matrix classes. For example, you might use statements like the following:

```
double func(Mat_I_DP &a, Vec_IO_DP &b);
{
    const int N=10;
    Mat_DP d(N,N);

    Vec_DP *e=new Vec_DP(N);
    ...

    delete e;
}
```

The complete set of these definitions is contained in the file `nrtypes.h`, reproduced in Appendix B. But you’ll find you seldom have to refer to it. In fact, if you haven’t already done so, now would be a good time to follow the instructions in the Quick Start subsection in §1.0. You’ll probably find you know enough to use the Recipes without any problems.

## A Few Wrinkles

We like to keep code compact, avoiding unnecessary spaces unless they add immediate clarity. We usually don't put space around the assignment operator “=”. For historical reasons, you will see us write `y= -10.0;` or `y=(-10.0);`, and `y= *a;` or `y>(*a);`. This is just because there used to be some C compilers recognize the (nonexistent) operator “=-” as being equivalent to the subtractive assignment operator “-=”, and “=\*” as being the same as the multiplicative assignment operator “\*=”. We hope that this quirkiness has disappeared by now, but we still have lingering habits.

We have the same viewpoint regarding unnecessary parentheses. You can't write (or read) C++ effectively unless you memorize its operator precedence and associativity rules. Please study the accompanying table while you brush your teeth every night.

We never use the `register` storage class specifier. Good optimizing compilers are quite sophisticated in making their own decisions about what to keep in registers, and the best choices are sometimes rather counter-intuitive.

We like to use the C++ constructor for casting types: `int(x)` rather than the C-style `(int) x`. Similarly, if a pointer to a function is passed as an argument, we invoke the function with the C++ form `func(x)` rather than the C-style `*func(x)`.

Some of our routines need to define a global vector or matrix to communicate data between two routines, let's call them `one()` and `two()`, without using function arguments. Typically the size of the data set needs to be set dynamically at runtime. We handle this situation by making the global variable a pointer to the data, so that function `one` would look something like this:

```
Vec_DP *xvec_p;           Definition of global pointer.

void one(...)
{
    ...
    xvec_p=new Vec_DP(n);  Allocate storage of size n.
    Vec_DP &xvec= *xvec_p; Make alias to simplify subsequent coding.
    ...
    delete xvec_p;       Reclaim storage when done.
```

The reference variable `xvec` is defined only to make subsequent code easier to write and read. Instead of writing `(*xvec_p)[i]` we can write `xvec[i]`.

To use the global vector in function `two`, we use the following scheme:

```
extern Vec_DP *xvec_p;    Declaration of global pointer, defined elsewhere.

void two(...)
{
    ...
    Vec_DP &xvec= *xvec_p; Make alias to simplify subsequent coding.
    ...
```

(Note for experts: An alternative scheme is to declare a zero-size vector globally, and then resize it appropriately inside function `one()`. However, we don't want to assume the existence of a `resize` function in the vector or matrix library.)

We have already alluded to the problem of computing small integer powers of numbers, most notably the square and cube. The omission of this operation

Operator Precedence and Associativity Rules in C++		
::	scope resolution	left-to-right
()	function call	left-to-right
[]	array element (subscripting)	
.	member selection	
->	member selection (by pointer)	
++	post increment	<b>right-to-left</b>
--	post decrement	
!	logical not	<b>right-to-left</b>
~	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	<b>right-to-left</b>
=	assignment operator	<b>right-to-left</b>
also += -= *= /= %=		
<<= >>= &= ^=  =		
,	sequential expression	left-to-right

from C++ is perhaps the language's most galling continuing insult to the scientific programmer. All good Fortran compilers recognize expressions like  $(a+b)**4$  and produce in-line code, in this case with only *one* add and *two* multiplies. It is typical for constant integer powers up to 12 to be thus recognized.

In `nrutil.h` we provide an inline templated function to handle squaring. Its definition is

```
template<class T>
inline const T SQR(const T a) {return a*a;}
```

You're on your own for higher powers. We also provide a collection of similar functions for other simple operations: SQR, MAX, MIN, SIGN, and SWAP. These do the obvious things (SIGN(a,b) returns the magnitude of a times the sign of b.)

Scientific programming in C++ may someday become a bed of roses; for now, watch out for the thorns!

#### CITED REFERENCES AND FURTHER READING:

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

## 1.3 Implementation of the Vector and Matrix Classes

In this section we describe the details of the implementation of the vector and matrix classes we use. We start with the default classes, and then describe how you can instead use another matrix/vector library with our Recipe functions. Unless you are an experienced C++ programmer, you will probably not want to read beyond the first subsection. And remember: in practice you will actually never need to use the names `NRVec` or `NRMat`. Instead, you use identifiers like `Vec_I_DP` as described in the previous section.

### The Default Vector and Matrix Classes

Here is the declaration of the `NRVec` class:

```
template <class T>
class NRVec {
private:
    int nn;                Size of array, indices 0..nn-1.
    T *v;                 Pointer to data array.
public:
    NRVec();              Default constructor.
    explicit NRVec(int n); Construct vector of size n.
    NRVec(const T &a, int n); Initialize to constant value a.
    NRVec(const T *a, int n); Initialize to values in C-style array a.
    NRVec(const NRVec& rhs); Copy constructor.
    NRVec& operator=(const NRVec& rhs); Assignment operator.
    NRVec& operator=(const T& a); Assign a to every element.
```

```

    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.
    ~NRVec();                                       Destructor.
};

```

The private variables are discussed in Appendix B. Let's look at the public interface. The various constructors allow vectors to be declared as in the following examples:

```

NRVec<double> v;           Zero-size array.
NRVec<double> w(10);      w[0..9].
NRVec<int> x(a,10);       x[0..9] = a.
NRVec<int> y(b,10);       y[0..9] = b[0..9], b a C-style array.
NRVec<int> z=y;           z[0..9] = y[0..9], y an NRVec.

```

The `explicit` keyword prevents the compiler from performing an implicit type conversion from an integer. Unless you know what you are doing, you almost certainly don't need to do such conversions, which are often a source of hard-to-diagnose errors. So using `explicit` is a good idea.

The overloaded assignment operators allow code like (assuming the declarations above)

```

y=x;           Copy values from x to y.
x=1;           Set all elements of x to 1.

```

The overloaded subscript operator `[]` is used repeatedly in the routines to access array elements. For example, `x[6]` is the seventh element in the vector. We will discuss the two forms of the subscript operator below in the subsection "More on `const` Correctness."

Finally, the `size()` function returns the number of elements in the vector:

```

NRVec<double> w(10);
int n = w.size();           Sets n to 10.

```

As we will discuss later, you can use any vector class you like with the *Numerical Recipes* functions as long as it provides the basic functions above. (The functions can be called something else; it's the *functionality* that must be the same.) In fact, you can get away with even less. As long as you provide the constructor for a vector of length `n`, the subscript operator, and the `size()` function, most of the Recipes will work. About ten will have to have some code replaced by explicit loops to handle initializing to values or arrays. (The *Numerical Recipes* example routines make more extensive use of the additional functions in the `NRVec` class.)

The matrix class `NRMat` is very similar to `NRVec`:

```

template <class T>
class NRMat {
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:
    NRMat();           Default constructor.
    NRMat(int n, int m); Construct n x m matrix.
    NRMat(const T& a, int n, int m); Initialize to constant value a.

```

```

NRMat(const T *a, int n, int m);           Initialize to values in C-style array a.
NRMat(const NRMat& rhs);                  Copy constructor.
NRMat& operator=(const NRMat& rhs);      Assignment operator.
NRMat& operator=(const T& a);           Assign a to every element.
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.
~NRMat();                                 Destructor.
};

```

The only point to note is the return type of `operator[]`. If `a` is of type `NRMat`, we want `a[i]` to point to row `i` of the matrix so that `a[i][j]` will be the  $(i, j)$  matrix element. In our default class shown above, this means that `a[i]` must be of type `T*`. However, it might be something else in a different, more complicated, class library.

Just as in the case of vectors, you will find out below how to use any matrix class with the Recipes, as long as it provides the basic functions above. And if all you supply is the constructor for an  $m \times n$  matrix, the subscript operator, and the functions for the number of rows and columns, only about five routines will need to be rewritten.

Only two of our routines use a *three*-dimensional array: `rlft3` in §12.5 and `solvde` in §17.3. This data structure is provided by the class `NRMat3d` in `nrutil.h`. We have not made any special efforts for you to be able to use your own class instead of the one we provide, but if necessary you could follow the same technique we describe below for vectors and matrices.

The full implementation code for the `NRVec` and `NRMat` classes is in `nrutil.h`, which is reproduced in Appendix B. Note that you can easily use the standard library class `valarray` for vectors instead of `NRVec`: Simply replace the complete declaration and implementation of the `NRVec` class with the following:

```

#define NRVec valarray
#include <valarray>

```

Alas, the standard library doesn't provide anything comparable to `valarray` for matrices.

### More on Const Correctness

At this point we need to elaborate on what exactly `const` does for a non-simple 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 data members are unchanged. But note that 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 `NRVec` argument `a`. To avoid unnecessary copying, we always pass arrays by reference. Consider the difference between declaring the argument of a function with and without `const`:

```

void f(NRVec<double> &a)    OR    void f(const NRVec<double> &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 declaration 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 `operator[]` to be `const`. This is why there are two versions of `operator[]` in the `NRVec` class,

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

The first form returns a reference to an element of a modifiable vector, while the second is for a nonmodifiable vector. The way these work in our example is that if argument `a` of function `f()` above is declared as `const` reference, then when `a` is dereferenced the second form of `operator[]` will be invoked. (It is the trailing `const` in the declaration that promises not to modify the object being dereferenced, and which must agree in `const`’ness with the argument.) Then the return type `const` allows expressions like `x = a[i]` but forbids expressions like `a[i] = 4` (`a[i]` is not an lvalue). By contrast, if `a` is declared in `f()` without the `const` qualifier, the first form of `operator[]` is invoked, which allows both kinds of statements (`a[i]` can be an lvalue).<sup>1</sup>

This is the kind of trickery you have to resort to to get `const` to protect the data. As judged from the large number of matrix/vector libraries that follow this scheme, many people feel that the payoff is worthwhile.

However, we must also recognize the existence of an alternative implementation of `operator[]`, which is to stick with the basic C++ philosophy “`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 the size and pointer of the vector are unchanged, and element `i` is returned as potentially modifiable.

While we do not use this second alternative in our default classes, since it is nice to be able to use `const` to protect the contents, we *must* use it if we replace our default classes with the classes described in the next subsection that allow you to use your own favorite matrix/vector library instead of our defaults. The reason is that with this alternative, all invocations of `operator[]` will enforce `const`ness of the object so as to allow the possibility of automatic conversions. (When an object is passed to a function by reference, automatic type conversions can be made only if the object is `const`.) Although giving up the extra `const` checking is regrettable,

<sup>1</sup> “Wait!” you might object. “The first form of `operator[]` doesn’t actually modify the object, only the data. Why can’t I put a trailing `const` after it?” Answer: because then the compiler could not distinguish between the two forms. Overloaded functions and operators must be distinguishable by the argument types alone, not by the return types.

this nevertheless turns out to be the best route to allowing transparent use of other matrix/vector libraries, as we now explain.

## Using Other Vector and Matrix Libraries

Suppose you have a vector class `MyVec` that you want to use instead of `NRVec`. The goal is to have automatic type conversions when a `MyVec` is passed to a Recipe function (and an `NRVec` is expected), and when an `NRVec` is returned by a Recipe function (and a `MyVec` is expected). C++ supplies us with the necessary tools: a “constructor conversion” to make an `NRVec` out of a `MyVec`, and an “operator conversion” to make a `MyVec` out of an `NRVec`.

The first key idea is to make `NRVec` be a “wrapper class” for `MyVec`, that is, an `NRVec` “holds-a” `MyVec`. The second key idea is to have the wrapper class hold both a `MyVec` *and* a reference to a `MyVec`. Then when we want to make an `NRVec` out of a `MyVec`, we can use the constructor initializer list to point the reference to the existing `MyVec`. No copying of data takes place at all.

The wrapper classes must provide all the functions of the default classes introduced earlier, and we shall describe them in the same order. So here is the start of the `NRVec` wrapper class:

```
template<class T>
class NRVec {
private:
    MyVec<T> myvec;
    MyVec<T> &myref;
public:
    NRVec<T>() : myvec(), myref(myvec) {}
    ...
}
```

The private variables are only a `MyVec` and a reference to one. The private variable `myvec` is used solely when creating a new `NRVec`. The default constructor just invokes the default constructor for a `MyVec`, and then points the reference accordingly. Similarly, the remaining constructors simply invoke the corresponding `MyVec` constructors:

```
explicit NRVec<T>(const int n) : myvec(n), myref(myvec) {}
NRVec<T>(const T &a, int n) : myvec(n,a), myref(myvec) {}
NRVec<T>(const T *a, int n) : myvec(n,a), myref(myvec) {}
```

Note we are assuming that the syntax of the `MyVec` class member functions is the same as that of the `NRVec` class, but it’s easy to take care of different syntax. For example, some vector classes expect arguments in the opposite order: `myvec(a,n)` instead of `myvec(n,a)`.

Next comes the conversion constructor. It makes a special `NRVec` that points to `MyVec`’s data, taking care of `MyVec` actual arguments passed as `NRVec` formal arguments in function calls:

```
NRVec<T>(MyVec<T> &rhs) : myref(rhs) {}
```

Since all the other functions in `NRVec` access the object only through `myref`, there is no need to initialize `myvec`.

The copy constructor and assignment operator simply call the `MyVec` copy constructor and assignment operator:

```
NRVec(const NRVec<T> &rhs) : myvec(rhs.myref), myref(myvec) {}
inline NRVec& operator=(const NRVec &rhs) { myref=rhs.myref; return *this;}
inline NRVec& operator=(const T &rhs) { myvec=rhs; return *this;}
```

(We assume here that the `MyVec` functions do the sensible thing of making a “deep” copy, that is, they copy the data, not just the reference. See the MTL example in Appendix B for a case where this is not true.)

Similarly, functions like `size()` are implemented by calling the corresponding `MyVec` functions:

```
inline int size() const {return myref.size();}
```

We pointed out earlier that for the subscript operator, only the form that guarantees constness of the container is allowed. We want automatic type conversion, and we get that only for objects passed by `const` reference. Accordingly, we cannot guarantee constness of the data (see the discussion in the previous subsection):

```
inline T & operator[](const int i) const {return myref[i];}
```

Next comes the conversion operator from `NRVec` to `MyVec`, which handles `NRVec` function return types when used in `MyVec` expressions:

```
inline operator MyVec<T>() const {return myref;}
```

Finally, the destructor is trivial: the `MyVec` destructor takes care of destroying the data.

```
~NRVec() {}
```

A wrapper class for `NRMat` follows exactly the same form as for `NRVec`. The only thing to watch out for is to make sure that the return type for `operator[]` is whatever a `MyMat` object returns for a single `[]` dereference. Then expressions like `a[i][j]` will work correctly.

Instead of listing the final form of the wrapper class here (pulling together all the lines above), we list in Appendix B two sample `nrutil.h` files for two popular matrix/vector libraries. They are for the Template Numerical Toolkit, or TNT [1], and the Matrix Template Library, or MTL [2]. We have found TNT to be particularly easy to use with the Recipe functions. All you need to do is use `TNT::Vector` for `MyVec` and `TNT::Matrix` for `MyMat`. If you are concerned about poor efficiency in using wrapper classes, our timing experiments show less than 10% overhead for TNT with most compilers, compared with using TNT directly.

One final instruction: the file `nrtypes.h` also needs to be changed to the “`const` protects the container, not the contents” convention for passing arguments by reference. To make this easy, we have supplied the file `nrtypes_lib.h` that correctly defines the constness of vector and matrix arguments in this way. So simply edit `nrtypes.h` to include this file instead of the default `nrtypes_nr.h` (see Appendix B).

A note for C++ aficionados: You can also implement the interface to other matrix/vector libraries by making `NRVec` a derived class of your vector class. However, this is not nearly as elegant as the wrapper class. In particular, it depends on the implementations inside of your vector class, while the wrapper class uses only the public interface and semantics of your vector class.

#### CITED REFERENCES AND FURTHER READING:

Pozo, R., *Template Numerical Toolkit*, <http://math.nist.gov/tnt>. [1]

Lumsdaine, A., and Siek, J. 1998, *The Matrix Template Library*, <http://www.lsc.nd.edu/research/mtl>. [2]