MATH Toolkit for REAL-TIME Programming

- Do big math on small machines
- Write fast and accurate library functions
- Master analytical and numerical calculus
- Perform numerical integration to any order
- Implement z-transform formulas

JACK W. CRENSHAW

Math Toolkit for Real-Time Programming

Jack W. Crenshaw

Cover art created by Janet Phares.

Distributed in the U.S. and Canada by:
Publishers Group West
1700 Fourth Street
Berkeley, CA 94710
1-800-788-3123

ISBN: 1-929629-09-5

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

Table of Contents

x Table of Contents

Supplementary Resources Disclaimer

Additional resources were previously made available for this title on CD. However, as CD has become a less accessible format, all resources have been moved to a more convenient online download option.

You can find these resources available here: www.routledge.com/9781138412477

Please note: Where this title mentions the associated disc, please use the downloadable resources instead.

Preface

Who This Book Is For

If you bought this book to learn about the latest methods for writing Java applets or to learn how to write your own VBx controls, take it back and get a refund. This book is not about Windows programming, at least not yet. Of all the programmers in the world, the great majority seem to be writing programs that do need Java applets and VBx controls. This book, however, is for the rest of us: that tiny percentage who write software for real-time, embedded systems. This is the software that keeps the world working, the airplanes flying, the cars motoring, and all the other machinery and electronic gadgets doing what they must do out in the real world.

Computers are for Computing

These days, computers seem to be everywhere and used for everything, from running the air conditioner in your car to passing money around the world in stock and commodities markets and bank transactions, to helping lonely hearts lover wannabes meet in Internet chat rooms, to entertaining Junior (or Mom and Dad) with ever more accurate and breathtaking graphics — some rated XXX. At one time, though, there was no doubt about what computers were for.

Computers were for computing. When Blaise Pascal and others designed their first mechanical calculators, it was to help them do arithmetic. When Charles Babbage designed the original computer in 1823, using mechanisms like cams, cogwheels, and shuttles, he didn't call it his Difference Engine for

nothing. In Babbage's day, the only way to perform numerical computations was by hand, and the worst part about computing things by hand is that people, being human, tend to make errors. An error partway through a long calculation ruins the rest and requires one to go back to the point of the error and start over. If the calculation takes on the order of a lifetime, you can see how the discovery of an error way back toward the beginning could ruin your whole day. It is hardly a surprise, then, that Babbage and his royal backers sought a faster and more reliable way of performing calculations. They didn't get it — the Babbage engine was never completed, but the dream remained alive.

The dream became more of an imperative during World War II. As usual, the team with the best weapons had the best chance of winning. Artillery firing tables took high priority, as did bombsights for airplanes. The Norden bombsight was perhaps the best kept secret of the war, next to The Bomb itself. Its miracles were accomplished through mechanical means, much like the difference engine conceived by Babbage, except that the Norden bombsight was an analog, not digital, computer. (The minor detail that it fit into an airplane also helped a lot.)

Researchers at Harvard, Yale, MIT, and other institutions recognized the value of computing things digitally and automatically. From their efforts came the original electric (i.e., relay) and electronic computers that showed up just at the end of the war. In the 1950s and 1960s, digital computers exploded on the scene, changing the world forever. Still, there was no doubt what those first computers were for. They were used to compute the trajectories of missiles, airplanes, and spacecraft. This was, in fact, the point where I entered the scene, calculating trajectories for NASA's Apollo missions, including the abort trajectories used for Apollo 13. Other people were using similar computers for more mundane, but still essential, functions, such as actuarial tables used by insurance companies. Computers were being used for computing, and this seemed quite proper at the time.

Shortly afterwards, however, a funny thing happened. In the process of computing difference equations and other tabular data, programmers found the need to do logic as well as arithmetic. If you're calculating something in a loop, you at least need to know when to quit. Often, this meant comparing two numbers and doing something different, depending on their values. The branch was born, which begat the `IF` test, which begat the `DO`, `FOR`, and `WHILE` loops.

Somewhere along the line, a bright person figured out, "Hey, if this thing can make decisions, maybe I can get it to make mine!" Someone surely must have envisioned a computer that could take all the world's data, be it

financial, political, or scientific, and reduce it to the ultimate degree: a simple output that said, "Buy Microsoft," or "Push the red button," or "Run!" The concept of non-numerical programming was born.

The concept gained a lot more momentum with the advent of systems programming tools like assemblers, compilers, and editors. You won't find much mathematics going on inside a C++ compiler or Microsoft Word, beyond perhaps counting line and column numbers. You certainly won't find any computing in Internet browsers. Today, the great majority of software running on computers is of the non-numeric type. Math seems to have almost been forgotten in the shuffle.

This tendency is reflected in the computer science curricula taught in universities. As is proper, most of these curricula stress the non-numeric uses of computers. Most do still pay at least lip service to numeric computing (today, it is called numerical analysis). However, it's possible at some universities to get a degree in Computer Science without taking one single course in math.

The problem is, despite the success of non-numeric computing, we still need the other kind. Airplanes still need to navigate to their destinations; those trig tables still need to be computed; spacecraft still need to be guided; and, sadly enough, we still need to know where the bombs are going to fall.

To this need for numeric computing has been added another extremely important one: control systems. Only a few decades ago, the very term, "control system" implied something mechanical, like the damper on a furnace, the governor on a steam engine, or the autopilot on an airplane. Mechanical sensors converted some quantity like temperature or direction into a movement, and this movement was used to steer the system back on track.

After World War II, mechanical cams, wheels, integrators, and the like were replaced by electronic analogs — vacuum tubes, resistors, and capacitors. The whole discipline of feedback theory led to a gadget called the operational amplifier that's still with us today and probably always will be. Until the 1970s or so, most control systems still relied on electronic analog parts. But during the late 1960s, aerospace companies, backed by the Defense Department, developed ruggedized minicomputers capable of withstanding the rigors of space and quietly inserted them in military aircraft and missiles.

Today, you see digital computers in places that were once the domain of analog components. Instead of using analog methods to effect the control, designers tend to measure the analog signal, convert it to digital, process it digitally, and convert it back again. The epitome of this conversion may well

lie hidden inside your CD disk player. The age of digital control is upon is. If you doubt it, look under the hood of your car.

As a result of the history of computing, we now have two distinct disciplines: the non-numeric computing, which represents by far the great majority of all computer applications, and the numeric computing, used in embedded systems. Most programmers do the first kind of programming, but the need is great, and getting greater, for people who can do the second kind. It's this second kind of programming that this book is all about. I'll be talking about embedded systems and the software that makes them go.

Most embedded systems require at least a little math. Some require it in great gobs of digital signal processing and numerical calculus. The thrust of this book, then, is twofold: it's about the software that goes into embedded systems and the math that lies behind the software.

About This Book

As many readers know, for the last five years or so I've been writing articles and a column for computer magazines, mostly for Miller Freeman's *Embedded Systems Programming* (ESP). My column, "Programmer's Toolbox," first appeared in the February 1992 issue of ESP (Vol. 5, #2) and has been appearing pretty much nonstop ever since. Although the title gives no hint, my emphasis has been on computer math, particularly the application of advanced methods to solve practical problems like simulation, analysis, and the control of processes via embedded systems.

Many of my articles are available on the CD-ROMs from ESP and *Software Development*. However, they're necessarily scattered across the disk, and it's not always easy to find the particular article one needs. Also, as I've drifted from subject to subject, sometimes in midcolumn, the seams between subjects have been more jangling than I might have liked.

For the last few years, I've been getting ever more insistent pleas from readers who would like to see all of my articles gathered together into a book, with the loose ends tied up and the joining seams smoothed over a bit. This book is the result.

Some of my articles have been almost exclusively of a programming nature; others, almost pure math with only the slightest hint of software. The first decision I had to make as plans for the book went forward was how, and whether, to separate the software from the math. Should I have two sections — one on software and one on math — or perhaps two volumes or even two separate books? Or should I try somehow to merge the two subjects? In the end, the latter decision won out. Although some topics

are much more math-intensive than others, they are all aimed at the same ultimate goal: to provide software tools to solve practical problems, mostly in real-time applications. Because some algorithms need more math than others, it may seem to you that math dominates some chapters, the software, others. But the general layout of each chapter is the same: First, present the problem, then the math that leads to a solution, then the software solution. You'll need at least a rudimentary knowledge of math and some simple arithmetic. But I'll be explaining the other concepts as I go.

Some readers may find the juxtaposition of theory and software confusing. Some may prefer to have the software collected all together into nice, useful black boxes. For those folks, the software is available alone on CD-ROM. However, I will caution you that you will be missing a lot, and perhaps even misusing the software, if you don't read the explanations accompanying the software in each chapter. I'm a firm believer in the notion that everything of any complexity needs a good explanation — even a simple-minded, overly detailed one. Nobody ever insulted me by explaining the obvious, but many have confused me by leaving things out. I find that most folks appreciate my attempts to explain even the seemingly obvious. Like me, they'd rather be told things they already know than not be told things they don't know.

On each topic of this book, I've worked hard to explain not only what the algorithms do, but where they came from and why they're written the way they are. An old saying goes something like, give a man a fish, and tomorrow he'll be back for another; give him a fishing pole, teach him how to use it, and he will be able to catch his own. I've tried hard to teach my readers how to fish. You can get canned software from dozens, if not thousands, of shrink-wrap software houses. You'll get precious little explanation as to how it works. By contrast, if my approach works and you understand my explanations, you'll not only get software you can use, but you'll also know how to modify it for new applications or to write your own tools for things I've left out.

About Programming Style

Since we're going to be talking about math and algorithms that will ultimately be used in computer software, it's inevitable that I show you programming examples. In those examples, you will not only see how the algorithms unfold, you will also get to see examples of my programming style. This is not a Bad Thing; a secondary purpose of this book is to teach programming style as it applies to embedded systems.

Needless to say, the style I'm most likely to teach is the one I use. Over the years, I've learned a lot of things about how and how not to write software. Some of it came from textbooks, but most came from trial and error. After some 40 years of writing software, I think I'm finally beginning to get the hang of it, and I think I have some things to teach.

In their seminal book, *Software Tools*, (Addison-Wesley, 1976) Brian Kernighan and P.J. Plauger (K & P) suggest that one of the reasons that some programmers — and experienced ones, at that — write bad programs is because no one's ever showed them examples of good ones. They said, "We don't think that it is possible to learn to program well by reading platitudes about good programming." Instead, their approach was to show by example, presenting useful software tools developed using good programming practices. They also presented quite a number of examples, taken from programming textbooks, on how *not* to write software. They showed what was wrong with these examples, and how to fix them. By doing so, they killed two birds with one stone: they added tools to everyone's toolboxes and also taught good programming practices.

This book is offered in the same spirit. The subject matter is different — K & P concentrated mostly on UNIX-like text filters and text processing, whereas we'll be talking about math algorithms and embedded software — but the principle is the same.

For those of you too young to remember, it may surprise you (or even shock you) to hear that there was a time when calling a subroutine was considered poor programming practice — it wasted clock cycles. I can remember being taken to task very strongly, by a professional FORTRAN programmer who took great offense at my programming style. He saw my heavy use of modularity as extravagantly wasteful. He'd intone, "180 microseconds per subroutine call." My response, "I can wait," did not amuse.

I was lucky; the fellow who taught me FORTRAN taught me how to write subroutines before he taught me how to write main programs. I learned modularity early on, because I couldn't do anything else.

In those days, I soon developed a library of functions for doing such things as vector and matrix math, function root-solving, etc. This was back when punched cards were the storage medium of choice, and the library went into a convenient desk drawer as card decks, marked with names like "Vectors" and "Rotation." In a way, I guess these decks constituted the original version of the Ada "package."

That library followed me wherever I went. When I changed jobs, the software I'd developed for a given employer naturally stayed behind, but

rest assured, a copy went along with me. I considered that library to be my toolbox, and I also considered it my duty to supply my own tools, in the same spirit that an automechanic or plumber is expected to bring his or her tools to the job. Many of those routines continue to follow me around, even today. The storage media are different, of course, and the languages have changed, but the idea hasn't changed. Good ideas never grow old.

A few years ago, I was asked to teach a class in FORTRAN as an adjunct professor. One look at the textbook that the school faculty had already chosen for the class told me that it was *not* a book I wanted to use. The examples were textbook examples right out of K & P's other seminal book, *Elements of Programming Style, 2nd edition* (McGraw-Hill, 1978), of how *not* to write FORTRAN. As is the case with so many textbooks, each program was just that: a stand alone program (and not a well-written one at that) for solving a particular problem. The issues of maintainability, reusability, modularity, etc., were not addressed at all. How could they be? Functions and subroutines weren't introduced until Chapter 9.

I decided to take a different tack, and taught the students how to write functions and subroutines on day one. Then I assigned projects involving small subroutines, beginning with trivial ones like `abs`, `min`, `max`, etc., and working up to more complex ones. Each student was supposed to keep a notebook containing the best solutions.

Each day, I'd ask someone to present their solution. Then we'd discuss the issues until we arrived at what we considered to be the best (I had more than one vote, of course), and that solution, not the students' original ones, was the one entered into everyone's notebook. The end result was, as in K & P's case, twofold. Not only did the students learn good programming practices like modularity and information hiding, they had a ready-made library of top-quality, useful routines to take to their first jobs. I blatantly admit that my goal was to create as many Crenshaw clones as possible. I don't know how well the school appreciated my efforts — I know they didn't ask me back — but the effect on the students had to be similar to the one the judge gets when they instruct the jury to forget they heard a particularly damaging piece of testimony. Try as they might, I suspect those students are still thinking in terms of small, reusable subroutines and functions. A little bit of Crenshaw now goes with each of them.

Do I hope to do the same with you? You bet I do. In giving you examples of math algorithms, it's inevitable that I show you a bit of the style I like to use. That's fine with me, and for good reason: the methods and styles have been thoroughly proven over many decades of practical software applications. You may already have a style of your own, and don't want to change

it. That's fine with me also. Nobody wants to turn you *all* into Crenshaw clones. If you take a look at my style, and decide yours is better, good for you. We all have to adopt styles that work for us, and I don't really expect the cloning to fully take. Nevertheless if, at the end of this book, I will have given you some new ideas or new perspectives on programming style, I will have done my job.

On Readability

One aspect of my style will, I hope, stand out early on: I tend to go for simplicity and readability as opposed to tricky code. I'm a strong believer in the KISS principle (Keep It Simple, Simon). When faced with the choice of using a tricky but efficient method, or a straightforward but slower method, I will almost always opt for simplicity over efficiency. After all, in these days of 800 MHz processors, I can always find a few more clock cycles, but programming time (and time spent fixing bugs in the tricky code) is harder to come by. Many people talk about efficiency in terms of CPU clock cycles, but unless I'm coding for an application that's going to be extremely time-critical, there's another thing I find far more important: the time it takes to bring a program from concept to reality. To quote K & P again, "First make it run, *then* make it run faster."

A corollary to this concept requires that programs be written in small, easily understandable and easily maintainable chunks. The importance of the cost of program maintenance is only now becoming truly realized. Unless you want to be maintaining your next program for the rest of your life, picking through obscure branches and case statements, and wondering why a certain construct is in there at all, you'll take my advice and write your programs, even embedded programs, in highly modular form.

About the Programs

When I first began the Toolbox column, my intent was to make it multilingual, presenting examples in Pascal, C, C++, Ada, and even more exotic languages. This turned out to be a totally impractical dream. All I succeeded in doing was confusing everyone. Worse yet, I ended up with tools that couldn't play together, because they were written in different languages.

In this book, I'll be sticking almost exclusively to C++. Don't try to read into this decision any great love for the language. There are many nice features of C++, and many others I absolutely detest. Come to think of it, though, the same is true of other languages. I have no great, favorite

language, though Pascal probably comes the closest. I find that the most rabid partisans for a certain language are those who have never (or rarely) programmed in any other. Once you've programmed a certain number of languages, writing in one or the other is no big deal, and the debates as to the pros and cons of each can become a little less heated. I'll be using C++ for my examples for quite a practical reason: It's the language that's most popular at the moment.

At the same time, I must admit that this choice is not without its downside. At this writing, few real-time embedded systems are being written in C++, and for good reason: the language is still in a state of flux, and some of its more exotic features present significant risk of performance problems. Over the last few years, C and C++ language expert P.J. Plauger has been engaged in an effort to define a subset of C++ for embedded systems. At this writing, that effort is complete; however, no major software vendor yet has a compiler available for that standard.

About the Author

I can't really say that I began with the computer age at its very beginning. I never worked with Admiral (then Captain) Grace Hopper, John von Neumann, Eckert and Mauchly, or the other greats who started the whole computer thing off. But I did manage to catch the very next wave. I wrote my first computer program in 1956 for an IBM 650 computer (we didn't actually have one, the professor had to desk-check our program). I studied lunar trajectories using an IBM 702, the forerunner of the 704/709x series. The programs were written in assembly language because that's all there was. No one had yet heard of FORTRAN, much less Pascal, C, or C++. I still remember, as though it was yesterday, seeing a fellow programmer writing something on a coding pad in a language that looked almost readable to an ordinary mortal. I asked him what he was doing. He said, "I'm writing a program in FORTRAN." That was my first awareness that compilers even existed.

When I graduated from college in 1959, I interviewed with, among others, a government agency called NACA. A few weeks later, I received an offer letter, on which someone had struck out the "C" with a ball-point pen and written an "S" above it. It's fair to say that I was at NASA from the beginning. I lived just down the road from Gus Grissom, who used to practice his scuba skills in the community swimming pool. Wally Schirra lived there too. Because of the astronaut's strange hours, we often passed each

other as I was driving to and from work, and we'd give each other a fellow-sports-car honk and wave.

I never really intended to be a computer programmer, much less a software engineer or computer scientist, terms that hadn't even been invented at the time. My formal training is in Physics, and I joined NASA to help put men on the Moon; as a sci-fi buff of long standing, I felt I could do nothing else. My early days at NASA involved nothing more complex than a Friden electric calculator and a slide rule. Few people who remember the early NASA projects — Mercury, Gemini, and Apollo — realize the extent to which it was all done without computers. True, we never could have guided the Eagle to the surface of the Moon without its flight computer and the software that went into it. In fact, many of the most significant developments in real-time software and digital guidance and control systems evolved out of that effort. But all the early work was done with slide rules, calculators, and hand-drawn graphs.

Over the years, I developed a collection of useful, general-purpose tools, which I carried around in a wooden card case, just as a pool shark carries his favorite cue stick. The box followed me from job to job, and I considered the contents the tools of my trade, just as real as those of a carpenter, mechanic, or plumber.

During my years in the space program, most of the work involved math-intensive computations. We were doing things like simulating the motion of spacecraft or their attitude control systems. We were learning as we went along, and making up algorithms along the way because we were going down roads nobody had trod before. Many of the numeric algorithms originally came from those developed by astronomers for hand computations. But we soon found that they were not always suitable for computers. For example, when an astronomer, computing by hand, came across special cases, such as division by a number near zero, he could see the problem coming and devise a work-around. Or when using numerical integration, a person computing by hand could choose a step size appropriate to the problem. The computer couldn't handle special considerations without being told how to recognize and deal with them. As a result, many of the algorithms came out remarkably different from their classical forms. Many new, and sometimes surprising, techniques also came out that had no classical counterparts. As I discovered and sometimes invented new algorithms, they found their way into that ever-growing toolbox of programs.

I was into microprocessors early on. When Fairchild came out with their first low-cost integrated circuits ($1.50 per flip-flop bit), I was dreaming of a home computer. Before Intel shook the world with its first microprocessor,

the 4004, I was writing code for a less general-purpose chip, but one still capable of becoming a computer.

In late 1974, MITS shook the world again with the advent of the $349, 8080-based Altair, the computer that *really* started the PC revolution, no matter what Steve Jobs says. By that time, I had already written floating-point software for its parent, the 8008. I wanted to catch this wave in the worst way, and I did, sort of. For a time, I managed a company that had one of the very first computer kits after the Altair. The first assembly language programs I wrote for hire were pretty ambitious: a real-time controller for a cold forge, written for the 4040, and a real-time guidance system, using floating-point math and a Kalman filter, to allow an 8080 to steer a communications satellite dish. Both worked and outperformed their requirements.

In 1975, before Bill Gates and Paul Allen had written Altair (later Microsoft) BASIC, we were building and selling real-time systems based on microprocessors, as well as a hobby computer kit. So why ain't I at least a millionaire, if not a billionaire? Answer: some luck, and a lot of hard work. You have to work hard, with great concentration, to snatch defeat from the jaws of victory, and that's exactly what my partners and I did. I expect we weren't alone.

I am relating this ancient history to emphasize that I think I bring to the table a unique perspective, colored by the learning process as the industry has developed and changed. Although I don't advocate that all other programmers of embedded systems go through the torture of those early days, I do believe that doing so has given me certain insights into and understanding of what's required to build successful, reliable, and efficient embedded systems. I'll be sharing those insights and understandings with you as I go. This will not be your ordinary how-to book, where the author gives canned solutions without explanation — solutions that you can either use or not, but cannot easily bend to your needs. Rather, I'll share alternative solutions and the trade-offs between them so that you can use my code and have the understanding necessary to generate your own, with a thorough knowledge of why some things work well and some don't. That knowledge comes from the experience I've gained in my travels.

Foundations

Chapter 1

Getting the Constants Right

It may seem strange that I'd spend any time at all discussing the proper way to define a numeric constant. After all, any beginning text on any programming language will tell you what constitutes a legal definition of a constant. Decimal integer constants consist of one to *N* decimal digits (where *N* varies from language to language); hex or binary numbers begin with `0x` or `0b`, and floating-point constants have a decimal point and possible exponent field. Hardly the kind of stuff to devote many words to. Yet, you'd be surprised how many computer programs end up broken or behaving erratically (which is functionally equivalent to being broken) because someone got a constant wrong. Part of the purpose of this chapter is to illustrate, as graphically as I can, how even the simplest concepts of programming can get you into trouble if not handled with care.

I assume you're advanced enough in your knowledge of programming to know that embedding constants as literals inside the code is generally a Bad Idea. One of the main reasons is that you run the risk of getting it wrong, either via a typo or simple sloppy programming, with multiple instances of the same constant. I once worked on the maintenance of a

FORTRAN program that had no fewer than five values for π, ranging from 12 digits down to, incredibly, three (3.14). This is clearly a Bad Idea, because if you assign a variable the value π in one place and test it in another place for equality to π, the test will likely fail. Likewise, if you're solving for an arctangent, whose output values are given as offsets from some submultiple of π, you're going to see discontinuities in the output of the function.

Because of the problems inherent in such things, programmers have been taught, almost from day one, to assign names to literal constants and use the names, rather than the literals, where they are needed. It goes without saying that such constants should also be correct and exact to a tolerance within the resolution of the computer word length.

I must stop at this point to say that, in my opinion, one can go way overboard on this issue of assigning names to constants. One would never (well, hardly ever) write:

```
#define     ZERO    0
#define     ONE     1
```

However, sometimes people get so carried away with the notion of avoiding embedded literal constants that they will go to extreme lengths, and the only effect is obfuscation of the results of an operation. When I see the statement

```
for(i = START_VALUE; i <= END_VALUE; i++)
```

I can see that I'm going to be doing something in a counted loop, but how many times? To find out, I must go track down START_VALUE and END_VALUE, which may well be in files separate from the one I'm working on — most likely in a C header file or, because nature is perverse, even in separate header files. This does nothing toward helping me to understand the program, much less maintain it. In cases where the numbers are not likely to change, the far simpler, and more transparent

```
for(i = 0; i <3; i++)
```

is much easier to understand. The lesson to be learned here is, do things properly, but don't be a slave to a convention just because it's conventional. Know when to use a literal and when not to. In general, the numbers that deserve to have names are either the long ones like π, that are hard to write,

or the ones that are likely to change in the future, like the size of data buffers. For obvious values like the dimensions of vectors in a 3-D universe, keep it simple and use the literal.

In any case, having established that at least some literal constants deserve to be named, the problem still remains how to best assign these names to them and, far more importantly, how to communicate their values to the places where they are needed. It may surprise you to learn that these questions have plagued programmers since the earliest days of FORTRAN. Back when all programs were monstrous, monolithic, in a single file, and in assembly language, there was no problem; in assembly, all labels are global, so duplicate names never occurred, and the value of the constant was universally accessible from anywhere in the program.

#define Constants to Avoid Common Errors

I said that one would "hardly ever" define a simple literal like 1. Is this always true? Sometimes it is useful to assign floating-point constants for common numbers such as 1, 2, or 3 by including the following statements in the header file.

```
#define One     1.0
#define Two     2.0
#define Three   3.0
```

It may seem frivolous to define a constant called One, and I suppose it is. The idea does have some merit, though. A common programming error made by beginners (and sometimes us old-timers, too) is to leave the decimal point out of a literal constant, resulting in statements such as

```
x = x + 1;
```

where x is a floating-point number. By doing this, you're inviting the compiler to generate a wasteful run-time conversion of the 1 from integer to float. Many compilers optimize such foolishness out, but some do not. I've found that most modern compilers let me use integer-like constants, like 0 and 1, and are smart enough to convert them to the right type. However, to take the chance is bad programming practice, so perhaps there's some value to setting up 0 and 1 as floating-point constants, just to avoid the mistake.

Sadly, some FORTRAN programmers continue to use monstrous, monolithic, single-file programs, which is hardly surprising when one considers

that this is the way the language is taught to them. In FORTRAN, however, programmers had, almost for the first time, the introduction of modularity via subroutines, and the principles of information hiding (not yet voiced as such by David Parnas, but still vaguely grasped as a Good Idea) prevented subroutines from having access to global data unless that access was explicitly granted. The mechanism for granting access to variables was to put them in a COMMON area. So, for example, I might write

```
COMMON PI, HALFPI, TWOPI,...
```

There was, however, a serious flaw in this mechanism. Early FORTRAN compilers provided no mechanism for assigning literal values to the constants. These constants were, in fact, not really constants, they were variables, and someone had to take responsibility for giving them values. Typically, the main program would have an initialization section, as shown in Listing 1.1.

Listing 1.1 Initializing constants in FORTRAN.

```
PI = 3.141592654
HALFPI = 1.570796327
TWOPI = 6.283185307
DPR = 57.29577951       {conversion from radians to degrees}
RPD = 1.74532952e-2     {and vice versa}
```

And so on. With a proper grasp of the concepts of modular programming, the main program might even call an initialization routine to take care of the initialization.

```
CALL CINIT
```

Related Constants

You might have noticed that the constants in the example are all related to each other and, ultimately, to π. One advantage of a FORTRAN initialization section or subroutine is that the assignment statements were truly assignment statements. That is, they represented executable code that could

include expressions. A better implementation of the initialization code is shown in Listing 1.2.

Listing 1.2 Computed constants in FORTRAN.

```
PI = 3.141592654
HALFPI = PI / 2.0
TWOPI = 2.0 * PI
DPR = 180.0 / PI
RPD = PI / 180.0
```

This code not only makes it quite clear what the variables are, but more importantly, it leaves me with only one constant to actually type, decreasing the risk of making an error. If I went to a machine with different precision, changing just the one number got me back in business. The only remaining problems with this approach were simply the bother of duplicating the `COMMON` statement in every subroutine that used the constants and remembering to call the initialization subroutine early on.

Recognizing the need to initialize variables, the designers of FORTRAN IV introduced the concept of the Block Data module. This was the one module in which assignment to `COMMON` variables was allowed, and it was guaranteed to set the values of all of them before any of the program was executed. This relieved the programmer of the need to call an initializer. FORTRAN IV also provided named common blocks, so I could create a single block named `CONST` with only a handful of constant values. (Earlier versions had just a single `COMMON` area, so the statement had to include every single global variable in the program.)

In one sense, however, the Block Data module represented a significant step backward. This was because the "assignment" statements in Block Data only looked like assignments. They weren't, really, and did not represent executable code. Therefore, the code in Listing 1.2 would not work in a FORTRAN IV Block Data module.

Time Marches On

Except for hard-core scientific programmers, FORTRAN is a dead language. However, the need to use named constants, which involves not only assigning them values but also computing certain constants based upon the values of others and making them available to the modules that need them, lives on. You might think that the designers of modern languages like C and C++ would have recognized this need and provided ample support for it,

but you'd be wrong. Most of us find ourselves settling on partial solutions that do not completely satisfy.

In C or C++, the obvious mechanism for defining a constant is the preprocessor #define command.

```
#define pi 3.141592654
```

You can put these definitions at the top of each C file that needs them, but that leaves you not much better off than before. Unless the project has only one such file, you must duplicate the lines in each file of your project. This still leaves you with multiple definitions of the constants; you've only moved them from the middle of each file to the top.

The obvious next step is to put all the definitions in a central place — namely, a header file, as in Listing 1.3. Note that I've put an #ifndef guard around the definitions. It is always good practice in C to avoid multiple definitions if one header file includes another. Always use this mechanism in your header files.

Note that I've reverted to separate literals for each constant. This is because the #define statement, like the FORTRAN Block Data assignments, doesn't really generate code. As I'm sure you know, the preprocessor merely substitutes the character strings inline before the compiler sees the code. If you put executable code into the #defines, you run the risk of generating new executable code everywhere the "constant" is referenced.

Listing 1.3 A header file for constants.

```
// file constant.h
// definitions for common constants

#ifndef CONSTANTS
    #define CONSTANTS

    #define PI                      3.141592654
    #define HALFPI                  1.570796327
    #define TWOPI                   6.283185307
    #define RADIANS_PER_DEGREE      1.74532952e-2
    #define DEGREES_PER_RADIAN      57.29577951
    #define ROOT_2                  1.414213562
    #define SIN_45                  0.707106781

#endif
```

For constants such as TWOPI, this is not such a big deal. If you write

```
#define HALFPI    (PI/2.0)
```

the expression will indeed be substituted into the code wherever HALFPI is used (note the use of the parentheses, which are essential when using the preprocessor in this way). However, most compilers are smart enough to recognize common, constant subexpressions, even with all optimizations turned off, so they will compute these constants at compile time, not run time, and execution speed will not be affected. Check the output of your own compiler before using this trick too heavily.

But consider the constant ROOT_2. I'd like to write the following statement.

```
#define ROOT_2 (sqrt(2.0))
```

However, this contains a call to a library subroutine. It is guaranteed to generate run-time code. What's more, you'd better include <math.h> in every file in which ROOT_2 is referenced, or you're going to get a compilation error.

Does Anyone Do It Right?

I'm dwelling on this business of defining constants simply to show you that, even for so simple a problem as defining π or its related constants, there are no simple, pat solutions, even in a language as advanced as C++. To my knowledge, only two languages give the mechanisms you really need to do it properly, and one of them is rather clumsy to use.

The Ada language allows you to separate code into packages. Each package can contain data global to the package, global to the world, or local. It can also include tasks and functions. Finally — and this is the secret that makes it so useful — a package can have a package body that is executed as the application is started. This provides a place to put the code to initialize global constants. The only problem here is that you must refer to the constant with the package prefix: constant.pi, for example.

Borland's Object Pascal, as used in Delphi, does it even better. Borland Pascal allows for the use of units, one of which can contain all the global constants. As in the Ada package, the Borland unit can have an executable part, which is guaranteed to run before the main program begins to execute. Listing 1.4 shows a total, permanent solution in Borland Pascal. As far as

I'm concerned, this is the ideal solution to the problem. The only catch is, you can only do it in Borland Pascal.

Listing 1.4 The Pascal solution.

```
Unit Constants;Interface
Const One   = 1.0;
Const Two   = 2.0;
Const Three = 3.0;
Const Four  = 4.0;
Const Five  = 5.0;
Const Ten   = 10.0;
Const Half   = One/Two;
Const Third  = One/Three;
Const Fourth = One/Four;
Const Fifth  = One/Five;
Const Tenth  = One/Ten;

{In Borland Pascal, Pi is predefined}
Const TwoPi              = Two * Pi;
Const Pi_Over_Two        = Pi/Two;
Const Pi_Over_Three      = Pi/Three;
Const Pi_Over_Four       = Pi/Four;
Const Degrees_Per_Radian = 180.0/Pi;
Const Radians_Per_Degree = Pi/180.0;

{These constants will be filled by startup code}
Const Root_Two:   real = 0.0;
Const Root_Three: real = 0.0;
Const Root_Five:  real = 0.0;
Const Golden:     real = 0.0;
Const Sin_45:     real = 0.0;
Const Sin_30      = Half;
Const Cos_30:     real = 0.0;

Implementation
```

Listing 1.4 The Pascal solution. (continued)

```
Begin
    Root_Two      := Sqrt(Two);
    Root_Three    := Sqrt(Three);
    Root_Five     := Sqrt(Five);
    Golden        := (Root_Five + One)/Two;
    Sin_45        := One/Root_Two;
    Cos_30        := Root_Three/Two;
End.
```

You could use the `Unit Constants`, for example, to create functions that convert degrees to radians. I can't remember how many lines of FORTRAN code I've seen devoted to the simple task of converting angles from degrees to radians and back.

```
PSI   = PSID/57.295780
THETA = THETAD/57.295780
PHI   = PHID/57.295708
```

Not only does it look messy, but you only need one typo (as I deliberately committed above) to mess things up.

The `Unit Constants` in Listing 1.4 allows you to create the following functions.

```
{ Convert angle from radians to degrees }
Function Degrees(A: Real): Real;
Begin
    Degrees := Degrees_Per_Radian * A;
End;

{ Convert angle from degrees to radians }
Function Radians(A: Real): Real;
Begin
    Radians := Radians_Per_Degree * A;
End;
```

To use them, you need only write the following.

```
Psi := Radians(PsiD);
```

If these two routines are in a module that uses constants, you can be sure that the conversion will always be correct to the appropriate degree of precision. Note that the function call has the look of a type cast, which is sort of what it is.

You might wince at the inefficiency of a single-line function call to do this job, but in this case it's perfectly justified. Remember, this kind of conversion is typically used only during input and output, so efficiency is not much of an issue.

A C++ Solution

Can I do anything with C++ objects to get the equivalent of the package and unit initializations? Yes and no. I can certainly define a class called Constants with one and only one instance of the class.

```
class Constants{

public
    double pi;
    double twopi;
    Constants( );
    ~Constants( );

} constant;
```

I can then put the code — any code — to initialize these variables into the class constructor. This solution is workable, but I am back to having to qualify each constant as I use it, just as in Ada: constant.pi. The syntax gets a bit awkward. Also, there must surely be some run-time overhead associated with having the constants inside a class object.

Fortunately, if you're using C++, you don't need to bother with either classes or preprocessor #defines. The reason is that, unlike C, C++ allows for executable code inside initializers. Listing 1.5 is my solution for making constants available in C++ programs. Notice that it's an executable file, a

.cpp file, not a header file. However, it has no executable code (except what's buried inside the initializers).

Listing 1.5 The C++ solution.

```
/* file const.cpp
 *
 * defines commonly used constants for use by other modules
 *
 */
double pi          = 3.141592654;
double halfpi      = pi / 2.0;
double twopi       = 2.0 * pi;
double radians_per_degree = pi/180.0;
double degrees_per_radian = 180.0/pi;
double root_2      = sqrt(2.0);
double sin_45      = root_2 / 2.0;
```

Don't forget that you must make the constants visible to the code that references it by including a header file (Listing 1.6).

Listing 1.6 The header file.

```
/* File constant.h
 *
 * header file for file constant.cpp
 */

#ifndef CONSTANT_H
    #define CONSTANT_H

extern double pi;
extern double halfpi;
extern double twopi;
extern double radians_per_degree;
extern double degrees_per_radian;
extern double root_2;
extern double sin_45;

#endif
```

This solution is as near to perfect as you're likely to get with C++ or any other language in the near future. It's my standard approach. There remains only one nagging problem: the issue of data precision. I've shown the "constants" as double-precision floating-point numbers, but you may prefer to define the variables as `float`s to save space (though I don't recommend it; in these days of multimegabyte memory, get reckless and use the extra 14 bytes). Some compilers also allow the type `long double`, which gives access to the 80-bit internal format of the Intel math coprocessor. This type naturally requires more storage, but again, who cares? More to the point, because `long double` is the natural format for the math coprocessor, moving data into and out of it is faster if no conversions are required, so you might actually get a faster program with the higher precision.

There is no way around it, though, mixing data types can be a real pain in the neck, because the compiler is going to flood you with warnings about implicit type conversions and potential loss of accuracy. My best advice is, pick one precision, use it for all the floating-point variables in your program, and write the `constant.cpp` file to match. If you do decide to go with `long double`, don't forget that you must call the math functions by different names — usually `sqrtl`, and so on. You also must put an `L` suffix on all literals.

Header File or Executable?

There's another approach to defining constants that avoids the need to write a separate header file and includes the `.cpp` file in the project make list. This is to put the statements in the header file, as shown in Listing 1.7.

Note that I now have executable code in the header file. This is usually a no-no, but in this case it seems reasonable. Depending on the compiler you use, you might get a warning message that the header file contains executable code, but the program will still work just fine.

Listing 1.7 Constants in header file.

```
/* File constant.h
 *
 * defines commonly used constants for use by other modules
 *
 */
```

Listing 1.7 Constants in header file. (continued)

```
#ifndef CONSTANT_H
#define CONSTANT_H

static double const pi                 = 3.141592654;
static double const halfpi             = pi / 2.0;
static double const twopi              = 2.0 * pi;
static double const radians_per_degree = pi/180.0;
static double const degrees_per_radian = 180.0/pi;
static double const root_2             = sqrt(2.0);
static double const sin_45             = root_2 / 2.0;

#endif
```

Think a moment about the implications of using the header file shown above. Remember, this file will be included in every source file that references it. This means that the variables `pi`, `halfpi`, and so on will be defined as different, local variables in each and every file. In other words, each source file will have its own local copy of each constant. That's the reason for the keyword `static`: to keep multiple copies of the constants from driving the linker crazy.

It may seem at first that having different local copies of the constants is a terrible idea. The whole point is to avoid multiple definitions that might be different. But in this case, they can't be different because they're all generated by the same header file. You're wasting the storage space of those multiple copies, which in this example amounts to 56 bytes per reference. If this bothers you, don't use the method, but you must admit, that's a small amount of memory.

The sole advantage of putting the constants in the header file is to avoid having to link `constant.obj` into every program, and by inference, to include it in the project make file. In essence, you're trading off memory space against convenience. I guess in the end the choice all depends on how many programs you build using the file and how many separate source files are in each. For my purposes, I'm content to put the constants in the `.cpp` file instead of the header file, and that's the way it's used in the library accompanying this book. Feel free, however, to use the header file approach if you prefer.

Gilding the Lily

In Listing 1.5, I managed to get the number of constants I must explicitly specify down to one: π. In the listing, I've fixed it to 10 digits. That's a nice number, but those are too many digits for type `float`, and not enough for type `double`. The first type is accurate only to six or seven digits and the latter to 12 digits. Things really begin to get interesting if I use `long double`, where I must specify π to 22 digits. Then it's time for a quick run to the handbook.

To get around this problem, there's an alternative approach, which Intel recommends. Remember that the math coprocessor stores a value of π internally; it must in order to solve for arctangents. Instead of trying to remember π to 22 digits and key it in, why not ask the coprocessor what value it recommends?

```
double pi    = 4.0 * atan(1.0);
```

Clearly, this will work regardless of the precision you choose, the word length of the processor, or anything else. It also has the great advantage that you are absolutely guaranteed not to have problems with discontinuities as you cross quadrant boundaries. You may think it is wasteful to use an arctangent function to define a constant, but remember, this is only done once during initialization of the program.

The code shown in this chapter is included in Appendix A, as is all other general-purpose code. Bear in mind that I have shown two versions of the file `constant.h`. You should use only one of them, either the one in Listing 1.6 or Listing 1.7.

Chapter 2

A Few Easy Pieces

As long as I'm building a math library, I'll start with some basic functions. In this process, you'll be following in the footsteps of my old FORTRAN students — but fear not: this is not some useless textbook exercise (for that matter, neither was theirs). You'll be using the functions given here extensively in what follows.

About Function Calls

Lately I've been bumping into, and therefore wondering about, some peculiar aspects of the definitions of some very fundamental math functions. The peculiarity revolves around two facts:

Fact 1 As I'm sure you know, all C library functions using floating-point variables are set up to use `double`, rather than mere `float`, variables for both input arguments and return values. All `float` (or integer, of course) variables are promoted to `double` when the function is called and demoted again, if necessary, on the way out.

A few years ago, this used to be a good argument for staying with FORTRAN to do number crunching. If the precision of `float` is good enough for your task, you don't want the compiler to burn unnecessary clock cycles in

type conversions at every single function call. Aside from the overhead of the type conversions, the higher precision required by `double` also makes the required series expansions longer than they need to be and the functions therefore significantly slower. Why did the authors of C decide to write the library using only `double` variables? I have no idea, but I have a theory:

- They didn't want to bother writing two versions of every function, one for `float` and one for `double` data types.
- They weren't primarily the kind of people who did a lot of number crunching, so they didn't care that they were burdening the language with unnecessarily slow performance.

Whatever the true reason, this peculiarity is one reason many C programs run at about half the speed of their FORTRAN counterparts. And that's one of the reasons why you still see scientists doing their serious programming in FORTRAN (the other is C's lack of conformant arrays; more on this later).

Some programming shops, committed to using floating point and C in real-time situations but needing all the speed they can get, have gone so far as to write their own math libraries, redefining all the math functions using `float` variables.

More recently, the pendulum has swung the other way. Many programmers are now using at least 486, if not Pentium, processors, even in real-time applications. Others are using Motorola processors like the 68xxx in similar applications. Both families include math coprocessors, either as options or built into the CPU chip. These math coprocessors typically have all the precision they need to do double-precision arithmetic with no noticeable degradation in performance.

The Intel math coprocessor uses an 80-bit format, which is usually supported (though not guaranteed to be) by the C type `long double`. In this situation, you'll get the most efficient code by defining all of your variables as `long double`. Suddenly, the C convention doesn't deliver too much precision — it doesn't deliver enough.

Fact 2 The C preprocessor supports macros with passed parameters. However, it doesn't know or care about variable types (which, I hasten to point out, means that you can abandon all hope of compile-time type

checking using macros). The fundamental math functions `min()` and `max()` are defined as macros in the header file `math.h`.

```
#define max(a,b)    (((a) > (b)) ? (a) : (b))
#define min(a,b)    (((a) < (b)) ? (a) : (b))
```

Because of this definition, I can use these "functions" for any numeric type. I'll be using these two functions heavily.

I must tell you that I'm no fan of the C preprocessor. In fact, I'm President, Founder, and sole member of the Society to Eliminate Preprocessor Commands in C (SEPCC). I suppose the idea that the preprocessor could generate pseudofunctions seemed like a good one at the time. In fact, as in the case of `min()`/`max()`, there are times when this type blindness can be downright advantageous — sort of a poor man's function overloading.

For those not intimately familiar with C++, one of its nicer features is that it allows you to use the same operators for different types of arguments, including user-defined types. You're used to seeing this done automatically for built-in types. The code fragment

```
x = y * z
```

works as expected, whether `x`, `y`, and `z` are integers or floating-point numbers. C++ allows you to extend this same capability to other types. However, as noted in the text, this only works if the compiler can figure out which operator you intended to use, and that can only be done if the arguments are of different types.

Operator overloading is perhaps my favorite feature of C++. However, it is also one of the main reasons C++ programs can have significant hidden inefficiencies.

As noted, you lose any hope of having the compiler catch type conversion errors. However, the alternatives can be even more horrid, as the next function illustrates.

The simple function `abs()`, seemingly similar to `min()` and `max()` in complexity, functionality, and usefulness, is not a macro but a true function. This means that it is type sensitive. To deal with the different types, the C founding fathers provided at least three, if not four, flavors of the function: `abs()`, `fabs()`, `labs()`, and `fabsl()`. (This latter name, with qualifying characters both front and rear, earns my award as the ugliest function name

in all C-dom, especially since the trailing letter is, in many fonts, indistinguishable from a one.)

Because `abs()` is such a simple function, it can easily be defined as a macro:

```
#define abs(x) (((x) < 0)? -(x): (x))
```

It could be used for all numeric types, but for some reason, the founding fathers elected not to do so. Could this be because zero really has an implied type? Should I use `0.0L`, for `long double`? Inquiring minds want to know. In any case, I've found that most modern compilers let me use integer-like constants, like `0` and `1`, and are smart enough to convert them to the right type. Perhaps the presence of a literal constant in `abs()` is the reason it was defined as a function instead of a macro. If so, the reasons are lost in the mists of time and are no longer applicable.

What's in a Name?

Although I detest preprocessor commands, I'm also no fan of the idea of multiple function names to perform the same function. As it happens, I'm also President, Founder, and sole member of the Society to Eliminate Function Name Prefixes and Suffixes (SEFNPS). The whole idea seems to me a coding mistake waiting to happen.

In a previous life writing FORTRAN programs, I learned that the type of a variable, and the return type of a function, was determined via a naming convention. Integer variables and functions had names beginning with characters `I` through `N`. All others were assumed to have floating-point values.

As a matter of fact, even before that, the very first version of FORTRAN (before FORTRAN II, there was no need to call it FORTRAN I) was so primitive that it required *every* function name to begin with an `F`; otherwise, it couldn't distinguish between a function call and a dimensioned variable. I suppose that this FORTRAN would have had `FIABS()`, `FFABS()`, and `FFABSL()`.

When double-precision `floats` came along in FORTRAN, we had the same problem as we have now with C. While we could declare variables either real or double precision, we had to use unique names to distinguish the functions and subroutines. Although you could declare variables either real or double precision, you had to use unique names to distinguish the functions and subroutines. I can't tell you how many times I've written math libraries for FORTRAN programs, only to have to create another copy with names

ending in D, or some similar mechanism, when the program migrated to double precision. I didn't like the idea then, and I don't like it still.

In C++ you can overload function names — a feature I consider one of the great advantages of the language — despite the fact that the compiler sometimes has trouble figuring out which function to call. This means, in particular, that a function such as `abs()` can have one name and work for all numeric types.

Unfortunately, this is not done in the standard C++ library. To assist in portability, C++ compilers use the same math libraries as C. You can always overload the function yourself, either by defining a new, overloaded function or by using a template (assuming you can find a compiler whose implementation of templates is stable).

Alternatively, you can always define your own macro for `abs()`, as in the example above. This approach works for both C and C++. As with other macros, you lose compile-time type checking, but you gain in simplicity and in portability from platform to platform. Or so I thought.

Recently, I was developing a C math library that included a routine to minimize the value of a function. Because of my innate distaste for multiple names and the potential for error that lurks there, I included in my library the macro definition for `abs()`. I thought I was making the process simpler and more rational. Unfortunately, things didn't work out that way. The function minimizer I was building, using Microsoft Visual C/C++, was failing to converge. It took a session with CodeView to reveal that the compiler was, for some reason, calling the integer library function for `abs()` despite my "overloaded" macro. The reason has never been fully explained; when I rebooted Windows 95 and tried again the next day, the darned thing worked as it should have in the first place. Chalk that one up to Microsoft strangeness. But the whole experience left me wondering what other pitfalls lurked in those library definitions.

Digging into things a bit more led me to the peculiar "broken symmetry" between `min()`/`max()` and `abs()`. The macros `min` and `max` are defined in `<stdlib.h>`. The `abs()` function, however, is defined in both `<stdlib.h>` and `<math.h>`. (To my knowledge, this is the only function defined in both header files. What happens if you include both of them? I think the first one prevails, because both files have `#ifdef` guards around the definitions.)

My confusion was compounded when I tried the same code on Borland's compiler and found that the two compilers behaved differently. If you use `min()` or `max()`, and fail to include `<stdlib.h>`, both the Microsoft and Borland compilers flag the error properly and refuse to link. However, if you try the same thing with `abs()`, both compilers will link, and each produces a

bogus executable file. The Microsoft compiler at least gives an error message, but it's only a warning, and the .exe file is still generated.

The Borland compiler gives nary an error message and compiles cleanly, but the .exe file is still wrong; the "function," when called, returns a zero result, regardless of its input. To make matters even more interesting, try changing the file name from *.c to *.cpp. The Borland compiler will link in the library function for abs() (the integer version), even though you have not declared it anywhere. To my knowledge, this is the only case where you can call an undeclared function and get away with it.

The Borland help file gives a clue to what's going on with abs(): It says that you are free to write a macro version of it, but you must first use the statement

```
#undef abs
```

to turn off the existing definition. Ironically, the Borland compiler doesn't really seem to care if I do this or not. If I include the macro, it's used properly whether I use the #undef or not. On the other hand, the #undef does seem to help the Microsoft compiler, which appears to use the macro reliably; without it, it seems to do so only on odd days of the month or when the moon is full. Go figure.

If there's a lesson to be learned from this, I think it's simply that you should not take *anything* for granted, even such seemingly simple and familiar functions as min(), max(), and abs(). If you have the slightest doubt, read the header files and be sure you know what function you're using. Also, check the compiler's output. Don't trust it to do the right thing. In any case, the tools in this book use the macro abs. You'd better use the #undef too, just to be safe and sure.

What's Your Sign?

Sometimes it's useful to be able to assign the sign of one variable to the magnitude of another. I find this happens quite often in dealing with trig functions, as with the following function, taken right from the FORTRAN standard library:

```
#define sign(x, y) (((y)<0)? (-(abs(x))): (abs(x)))
```

Note the syntax: the value returned has the magnitude of x and the sign of y. Sometimes, you simply want to return a constant, such as 1, with the sign

set according to the sign of some other variable. No problem; just call `sign()` with `x = 1`.

The Modulo Function

There's another simple function that I use a lot, and it also comes straight from FORTRAN. It's called the modulo function [`mod()` in FORTRAN]. C/C++ has a similar function in the operator `%`, as in

```
x = y % z;
```

In theory, this function is supposed to return the remainder of `y`, after it's been divided by `z`. This C/C++, however, only works for integer arguments, while the `mod()` function is useful for `floats` as well. A handy usage is to reduce an angle to the range 0° to 360° degrees before doing anything else with it.

This construct is supposed to limit the first argument to a range given by the second. Thus, for example,

```
mod (1, 4) = 1
mod (7, 2) = 1
mod (5, 3) = 2
mod (4, 2) = 0
```

and so on. In other words, the result of `mod(x, y)` must always be between `0` and `y`.

However, if you look in the appropriate language manuals, you'll see both `mod()` and `%` are defined by the following code.

```
int n = (int)(x/y);
x -= y * n;
```

Unfortunately, this code doesn't implement the rules we'd like to have. Consider, for example, the following expression.

```
-5 % 2
```

By the preceding definitions, the result should be between `0` and `2`; it should, in fact, be `1`. Try it, however, on any C compiler and you'll get `-1`. Thus, the

result lies outside the range 0 to y, which to me seems a violation of the definition. The FORTRAN mod() works exactly the same way.

Is this a bug? That depends. If you use the definition given in the two lines of code above, which is the way the function is usually defined, it's not a bug. This is precisely the result one would expect from those two lines. If, however, you want the function to work like the mathematical definition of the modulo function, it's incorrect, since the latter requires that the result be bounded by 0 and whatever the second argument is. To me, that's a bug.

I use mod() for a lot of purposes, but one of them is to place angles in the proper quadrant. If I tell the computer to give me an angle between 0° and 360°, I dang well don't expect to see it coming back with -20°. I've struggled with this problem ever since my very first FORTRAN program, circa 1960. My solution has always been the same: write a replacement for the library function. Listing 2.1 is a mod() that works correctly for all arguments.

Listing 2.1 The mod() function.

```
// Return a number modulo the second argument
double mod(double x, double y){
    if(y == 0) return x;
    long i = (long)(x/y);
    if(x*y < 0)--i;
    x = x-((double)i)*y;
    if(x==y)x -= y;
    return x;
}
```

Chapter 3

Dealing with Errors

Every high-order language worthy of the name includes a library of arithmetic functions like `abs()`, `sqrt()`, and so on. Having those functions available, instead of having to write your own, is one of the very significant advantages of programming in a high-order language like C. In a moment, I'll discuss what you must do if such functions aren't available and you must roll your own. But for now, the good news is that any high-order language provides a ready-made library of fundamental functions. The bad news is, they can't be trusted.

The reason is simple enough: historically, these functions were designed to be fast at all costs. Error checking tends to be minimal to nonexistent. The programmer is expected to make sure that the arguments passed to the functions are such that the functions will work.

Consider the lowly square root function. The square of a real number is always positive, so the square root of a negative number is undefined. Try calling the square root function with a negative argument, and the program will abort with an error message like

```
Runtime error 207 at 0B74:00CA
```

That's hardly a helpful error message in the best of circumstances. About all it tells you is that a floating-point error occurred somewhere. To the programmer, the message is, "back to the debugger." To the average end user, it means, "call the vendor."

But consider the implications for, say, the manager of a complex and expensive chemical processing plant whose plant just shut down in midprocess leaving a large batch of half-cooked chemicals still reacting. In this case, the message means, "call your lawyer, then run like hell."

In Project Apollo, the landings of the Lunar Excursion Module (LEM) were computer assisted. Anyone who's ever played the old computer game Lunar Lander knows that the thrust schedule for descending and landing with zero velocity is tricky and best left to a computer. During that first historic landing of Apollo 11, astronauts Neil Armstrong and Buzz Aldrin depended on the computer to get them down. But when they were still rocketing toward the lunar surface, 500 feet up, still falling at great speed, and decelerating under a very precisely designed thrust program, they were suddenly greeted by the cheerful message

```
Alarm 1201
```

followed shortly by

```
Alarm 1202
```

Roughly translated, this means, "I've just gone out to lunch. You're on your own, kid." Having all the right stuff, Armstrong stoically took over and landed the LEM manually. The astronauts never voiced their reactions, but I know how I'd feel on learning that my transportation seemed to be failing, with the nearest service station 240,000 miles away. It's probably very much the same reaction that you'd get from that chemical plant manager. If I'd been them, the first person I'd want to visit, once I got home, would be the programmer who wrote that code.

The need for error checking in real-time embedded systems is very different than that for batch programming. The error mechanisms in most library functions have a legacy to the days of off-line batch programs, in which the system manager mentality prevailed; that is, "The programmer screwed up. Kick him off the machine and make room for the next one."

An embedded system, on the other hand, should keep running at all costs. That chemical plant or lunar lander must keep working, no matter what. It's not surprising, then, that the error-handling behavior of the standard library

functions should often need help for embedded applications. Any good embedded systems programmer knows better than to trust the library functions blindly. Quite often, you'll find yourself intercepting the errors before they get to the library functions.

Appropriate Responses

If a global halt is an inappropriate response in an embedded system, what is the appropriate one? That depends on the application. Usually, it means flagging the error and continuing with some reasonable value for the function result. Surprisingly, it often means doing almost nothing at all.

To show you what I mean, I will begin with floating-point arithmetic. The important thing to remember is that floating-point numbers are, by their very nature, approximations. It is impossible to represent most numbers exactly in floating-point form. The results of floating-point computations aren't always exactly as you expect. Numbers that are supposed to be `1.0` end up as `0.9999999`. Numbers that are supposed to be `0.0` end up as `1.0e-13`.

Now, zero times any number is still zero. Zero times zero is still zero. But `1.0e-27` times `1.0e-27` is *underflow*.

Almost all computers and coprocessors detect and report (often via interrupt) the underflow condition. When using PC-based FORTRAN, my programs occasionally halted on such a condition. This is not acceptable in embedded systems programming.

So what can you do? Simple. For all practical purposes, anything too small to represent is zero in my book. So a reasonable response is simply to replace the number with `0` and keep right on going. No error message is even needed.

Many compilers take care of underflows automatically. Others don't. If yours doesn't, you need to arrange an interrupt handler to trap the error and fix it.

Tom Ochs, a friend and outspoken critic of floating-point arithmetic, would chide me for being so sloppy and glossing over potential errors. In some cases, perhaps he's right, but in most, it's the correct response. If this offends your sensibilities, as it does Tom's, it's OK to send an error message somewhere, but be aware that it could cause your customer to worry unnecessarily. If you have a special case where underflow really does indicate an error, your program needs special treatment. But for a library routine that's going to be used in a general-purpose fashion, stick with the simple solution.