# FreePascal
From
Square One

Learn Pascal
programming from
the beginning...
the *real* beginning...
with no previous
programming
experience
required!

By Jeff Duntemann

Author of *Complete Turbo Pascal,*
Turbo Pascal Solutions,
Borland Pascal 7 from Square One, and
Assembly Language Step By Step
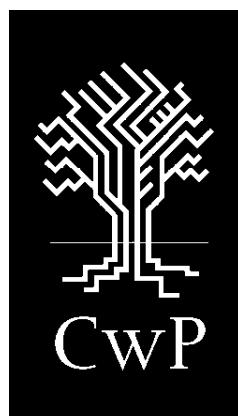
Copperwood Press

FreePascal From Square One

Volume 1:
The Fundamental Ideas of Programming
Installing and Configuring FreePascal and Lazarus
The Core of the Pascal Language

by Jeff Duntemann

Revision of 5/7/2025

Repeat...
Until...

FreePascal from Square One
By Jeff Duntemann

❧

Note that although I consider this book to be complete, there may be typos and other little glitches yet to be fixed. I work on it as time allows, and upload revisions as they happen. Check back now and then to see if there's a newer revision.

Table of Contents

Introduction:
How This Book Came to Be,
and
What I'm Trying to Do

Yessir, the book you're reading has been around the block a few times since I began writing it in November of 1983—which I boggle to think was over forty years ago at this revision. It's been through four print editions and on paper sold over 125,000 copies. It's been translated into five languages.

Now it's time set it free.

Pascal was not my first personal computer programming language. That honor belongs to FORTH, though I don't admit it very often. FORTH struck a spark in my imagination, but it burned like a pile of shredded rubber tires, leaving a stink and a greasy residue behind that took years to get rid of. BASIC came next, and burned like dry pine; with fury and small explosions of insight, but there was more light than heat, and it burned out quickly. BASIC taught me a lot, but it didn't take me far enough, and left me cold halfway into the night.

Pascal, when I found it, caught and burned like well-seasoned ash: Slow, deep, *hot*; I learned it with a measured cadence by which one fact built on all those before it. 1981 is a long time gone, and the fire's still burning. I've learned a lot of other languages in the meantime, including C, which burns like sticks of dynamite: It either digs your ditch or blows you to Kingdom Come. Or both. But when something needs doing, I always come back to Pascal.

When I began writing *Pascal From Square One* in the fall of 1983, I had no particular compiler in mind. There were several, and they were all (by today's standards) agonizingly bad. The best of the bunch was the long-extinct Pascal/MT+, and I used that as the host compiler for all of my example code. The book, however, was about Pascal the *language*: How to get started using it; how to think in Pascal; how to see the language from a height and then learn its component parts one by one.

When I turned the book in at the end of summer 1984, my editor at Scott, Foresman had a radical suggestion: Rewrite the book to focus not on Pascal the language but on Turbo Pascal the *product*. The maverick compiler from Scotts Valley was rapidly plowing all the other Pascals into the soil, and he smelled a new and ravenous audience. I took the manuscript back, and by January of 1985 I had

rewritten it heavily to take into account the numerous extensions and peccadilloes of Borland's Turbo Pascal compiler, version 2.0.

The book was delayed getting into print for several months, and as it happened, *Complete Turbo Pascal* was not quite the first book on the shelves focusing on Turbo Pascal. It was certainly the second, however, and it sold over 125,000 copies in the four print editions that were published between 1985 and 1993. The Second Edition of *Complete Turbo Pascal* ("2E", as we publishing insiders called it) came out in 1986, and addressed Turbo Pascal V3.0.

By March 1987 I had moved to Scotts Valley, California and was working for Borland itself, creating a programmer's magazine that eventually appeared in the fall of 1987 as *Turbo Technix*. Doing the magazine was a handful, and I gladly let others write the books for Turbo Pascal 4.0. I had the insider's knowledge that V5.0 would come soon enough, and change plenty, so I worked a little bit ahead, and *Complete Turbo Pascal, Third Edition* was published in early 1989.

It's tough to stay ahead in this business. Turbo Pascal 5.5 appeared in May of 1989, largely as a response to Microsoft's totally unexpected (but now long-forgotten) QuickPascal. V5.5 brought with it the new dazzle of object-oriented programming, and while I did write the V5.5 *OOP Guide* manual that Borland published with the V5.5 product I chose to pass on updating *Complete Turbo Pascal* for either V5.5 or V6.0.

The magazine *Turbo Technix* folded after only a year, and I left Borland at the end of 1988, wrote documentation for them until the end of 1989, and moved to Arizona in February of 1990 to start my own publishing company. This included my own programmers' magazine, *PC Techniques,* which became *Visual Developer* in 1996 and ran until early 2000.

The early 1990s saw some turbulence in the Pascal business. Borland retired the "Turbo Pascal" brand and renamed their flagship product Borland Pascal. The computer books division of my publisher, Scott Foresman, was engulfed and devoured by some other publisher. *Complete Turbo Pascal* was put out of print, and I got the rights back. An inquiry from Bantam Books in 1992 led to my expanding *Complete Turbo Pascal* into a new book focused on Borland Pascal 7. I was able to re-title the book *Borland Pascal 7 From Square One* (the title *Complete Turbo Pascal* had been imposed by Scott, Foresman) and bring it up to date. That edition didn't sell well because by 1994, Borland had created Delphi, and most of us considered the DOS era closed. I began writing books about Delphi and never looked back.

And so it was for almost 25 years. I considered just posting the word-processor files from *Borland Pascal from Square One* back in the 90s, but figured that no one was using Pascal all by itself anymore. Not so. I first saw FreePascal in 1998 and tried it from time to time, but it wasn't until January 2008 that Anthony Henry emailed me

to suggest that FreePascal could use a good tutorial, and it shouldn't be too hard to update *Borland Pascal 7 from Square One* to that role. He was right—and the project has been a lot of fun.

I mention all this history because I want people to understand where *FreePascal from Square One* came from, emphasizing that it's been a work in progress for forty-odd years. Why stop now? I don't have to cater to publishers, paper costs, print runs, or release schedules anymore. The book can evolve with the compiler, and every so often you can download the latest update. My publishing company Copperwood Press (will soon) offer a print-on-demand paper edition if you'd like one, but you can print it yourself if you prefer. That's how technical publishing ought to be, and someday will be.

What I'm Trying to Do Here

I wrote this book for ordinary people who had the itch to try programming, and with some dedication get good at it over time. You don't have to know anything at all about Pascal, or programming generally, to read it. Anyone who lives a tolerably comfortable life in this frenetic twenty-first century can program. Most of us (as I'll show a little later) engage the skills of programming to organize our daily lives. *If you want to program, you can.*

It's that simple.

And this book begins at what I call Square One: The absolute beginning. I will explain the concepts behind programming, how to install FreePascal and the Lazarus IDE, and how to craft simple text-based programs in the Pascal language. I will not cover programming GUI apps using the Lazarus GUI builder, which deserves an entire book of its own—one that I'm working on. You *do* need to know your way around your own computer and its operating system, but that's for housekeeping more than programming. This first volume will not get into operating system specific issues except with respect to installation. Part of FreePascal's magic is its portability: A simple program written under Windows can be recompiled without changes under Linux— or, for that matter, under any operating system to which FreePascal has been ported.

Rather than try to cover all of FreePascal in one book and do justice to none of it, I'm going to take my time to help you get the basics down cold. It's tough to build a fancy house when half the bricks are missing from your foundation. This whole book is about foundations, and getting familiar with all the skills that you will be using for the rest of your programming career, even (or especially) once you've forgotten how utterly fundamental they are.

That's why I call it Square One.

'80s Programming Style?
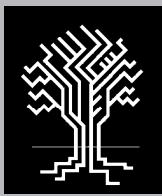
Some early readers objected to the coding style in this book's examples. To them it feels very 1980s. Yes, it does. That was a deliberate choice: Until windowed environments (Windows, Mac, modern Linux) became universal, that's how we programmed. GUI programming is a complicated business, and I can't even begin to discuss it in this first introductory volume. So the simplest way to show Pascal code in action is the fully textual command-line style, using read/readln and write/ writeln. I'm scoping out a brand new book that will begin with object-oriented programming (OOP) and jump from there into GUIs. Please bear with me. This is, I repeat, a book for absolute beginners. For their sake, I have to take it slow.

Why This Book Is Laid Out the Way It Is

I've been working on this book for a long time. I wanted it to serve equally well in two formats: print and ebook. This was a lot harder to do than I expected. Print, no problem. I used to lay out technical books for a living. Ebooks were a problem. People read ebooks on devices from the size of smartphones up to huge 32" monitors on their PCs. Today's ebooks finesse the problem by being "reflowable." Change the screen size, and the text resizes and reflows to match the screen.

I experimented with the major reflowable formats, and to be honest, they all looked awful. Books that are primarily text (fiction, or mostly textual nonfiction) look fine, and reflowing them works well. Once you introduce screenshots, tables, and code listings into the mix, the book descends into chaos if you reflow it away from the format in which it was originally laid out. And if reflowing it makes it unreadable, I figured it would be better to simply leave it in the standard page-image PDF format. This required using a larger font than most technical books use, as a compromise between paper and smaller displays like the IPad and Galaxy Tabs. I chose the A4 page size in part because I thought that most people who would be interested in the book lived outside the US, and in part because it maps a little more closely to the typical non-iPad wide-format 9:16 tablet display.

As always, I welcome suggestions and ideas, as well as feedback on typos, errors and obsolete information.

Part I:
The Fundamental Ideas
of Programming

CHAPTER 1.
THE BOX
THAT FOLLOWS A PLAN

There is a rare class of human being with a gift for aphorism, which is the ability to capture the essence of a thing in only a few words. My old man was one; he could speak volumes in saying things like, "Kick ass. Just don't miss." Lacking the gene for aphorism, I write books—but I envy the ability all the same.

The patron aphorist of computing is Ted Nelson, the eccentric wizard who created the perpetually almost-there Xanadu database system, and who wrote the seminal book *Computer Lib/Dream Machines*. It's a scary, wonderful work, in and out of print for forty years but worth hunting down if you can find it. In six words, Ted Nelson defined "computer" better than anyone else probably ever will:

A computer is a box that follows a plan.

We'll come back to the box. For now, let's think about plans, where they come from, and what we do with them.

1.1. ANOTHER SCOTTSDALE SATURDAY MORNING

Don't be fooled. The world is virtually swimming in petroleum. What we need more of is...*Saturdays.*

It's 5:30 AM on the start of a Scottsdale weekend: The neighborhood woodpecker hammers on our tin chimney cap, loud enough to wake the dead, but just long enough to wake the sleeping. QBit stretches and climbs on my chest, wagging furiously as though to say, Hey, guy, time waits for no man. *Shake it!*

Over coffee and scrambled eggs, I sit down at the kitchen table with a quadrille pad and try to figure out how to cram thirty hours' worth of doing into a sixteen-hour day. The toughest part comes first: Simply remembering what needs to be done. (This grows harder once you get into your sixties. Trust me.) I brainstorm a list in pure stream-of-consciousness fashion, jotting things down in no particular order other than how they occur to me, hoping that when I'm done it's all there:

Read email.

Pay bills.

Put money into checking account if necessary.

Get cash at ATM.

Go to Home Depot.

    Get Thompson's Water Seal.

    Get 2 4' flourescent bulbs.

    Get more steel wool if we're out.

    Get more #120 sandpaper if we're out.

Get birthday present for Brian. Old West Books? He likes ghost towns.

Put together grocery list. Go to Safeway for groceries.

Sand rough spots on the deck.

Water seal the deck.

See if my back-ordered water filter cartridge is in at Sears; call first.

Replace refrigerator water filter cartridge if they have it.

Take the empty grill propane tank in and get a full one.

Do what laundry needs doing.

Go home and swim fifty laps.

That's a lot to ask of one day. If it's going to be done, it's going to have to be done *smart,* which is to say, in an efficient order so that as little time and energy gets lost in the process as possible. In other words, to get the most out of a day, you'd better have a *plan.*

Time and space priorities

In lining things up for the merciless sprint through a too-short day, you have to be mindful of both time and space. Time is (as always) crucial. Some things have to happen before other things. Some things have to happen before a certain time of the day, or within a time "window"—such as the business hours of a store you need to get to.

And on any mad dash through a metropolitan area the size of Phoenix and its many suburbs, space becomes an overwhelming consideration. You can't just go to one place, come home, then go to the next place, then come home, and go to another place, then come home again; not if each destination lies ten or twelve miles or more from home. You have to think about where everything is, and visit

all destinations in an order that minimizes needless travel—especially with gas prices at historical highs. Home Depot is on the way to Old West Books, so it makes sense to visit one on the way to the other. Hard decisions sometimes happen: Desert Flower Appliances is a long way off, and not along any convenient connect-the-dots path. Do you need to go there at all? If so, make sure you have to go—call first—and consider that the water tastes awful, yet you've been avoiding the trip for a couple of months.

Furthermore, there are always hard-to-define necessities that influence how and in what order you do things. In an Arizona summer, you simply *must* do grocery shopping dead last, and preferably at a store close to home, if you expect to keep the ice cream solid and the Tater Tots frozen. Faced with the 45°C summer highs we generally have for three months here in Scottsdale, most car air conditioners would at best keep you alive.

Finally, when pressed, most of us will admit that we rarely manage to get everything done on the day we intend to do it. Some things end up squeezed out of a too-tight day like watermelon seeds from between greasy fingers. Whether we realize it or not, we often schedule the least necessary things last, so that if we don't get to them it's no disaster. After all, tomorrow is another day, and the undone items will just head up tomorrow's (or next Saturday's) errands list.

One way or another, mostly on instinct but with some rational thought, we put together a plan. After mulling it for a few minutes, I took my earlier stream-of-consciousness errands list and drafted my actual plan as shown below:

Pay bills.
Call and add money to checking account if required based on balance.
Check in the garage for steel wool and #120 sandpaper.
Put together grocery list.
Call Desert Flower Appliances to see if the filters are in.
Check oil in the Durango. Add if necessary.
Run errands:
    Go out Shea Boulevard to hit Home Depot first.
        Get Thompson's Water Seal.
        Get 2 4' flourescent bulbs.
        Get more steel wool if we're out.
        Get more #120 sandpaper if we're out.
        Swap out empty propane tank.

Head west out Mountain View to Old West Books.

Buy Arizona ghost-towns book for Brian.

If water filters are in, go across town on Shea to Tatum Blvd. then north to Desert Flower Appliances. Buy water filters.

Go down Tatum Safeway for groceries.

Get cash at ATM.

Come home.

Replace water filter cartridge behind refrigerator. (Finally!)

Sand rough spots on the deck.

Water seal the deck.

Do whatever laundry needs doing.

Read EMAIL.

Swim fifty laps.

Collapse!

From a height and in detail

The plan (for it is a plan) just outlined was executed pretty much the way I wrote it. I added a few things I hadn't thought of the first time, like checking the oil in the Durango. The discipline of drafting a plan will often bring out details and issues that didn't come through the first time.

Much of the actual detail of the plan acted simply as a memory jogger. In the heat of the moment (or in the heat of a summer afternoon, desperate to get someplace—anyplace—air conditioned!) details often get forgotten. I knew, for example, why I wanted to go to Old West Books—to buy a book for my nephew's birthday. I was unlikely to forget that. But in bad traffic, or if the Durango had acted up, well, forgetfulness happens...

To be safe, I wrote the plan in more detail than I might have needed. I've lived here for years and know my way around the area pretty well, but there are those 60s moments sometimes when you forget that it's a long haul across on Shea to the appliances store.

A plan can exist, however, at various levels of detail. Had I more faith in my memory (or if I'd been stuck with a smaller piece of paper) I might have condensed some of the items above into summaries that identify them without actually describing them in detail. A less verbose but no less complete form of the plan might look like this:

Pay bills.

Replenish checking account from savings.

Put together grocery list.

Put together a Home Depot list.

Call Desert Flower Appliances to see if the filters are in.

Check in the garage for steel wool and #120 sandpaper.

Check oil in the Durango. Add if necessary.

Run errands:

Home Depot.

Old West Books.

Desert Flower Appliances.

Safeway.

Sand and water seal the deck.

If I get it, replace water filter cartridge behind refrigerator.

Do what laundry needs doing.

Read EMAIL.

Swim fifty laps.

Collapse!

Look carefully at the differences between this list and the previous list. Mostly I've condensed obvious, common-sense things that I was unlikely to forget. I've been to the various stores so often that I could do it asleep, so there's really little point in giving myself driving directions. I have an intuitive grasp of where all the stores are located, and I can plot a minimal course among them all without really thinking about it. At best, the order in which the stores are written down jogs my memory in the direction of that optimal course.

I combined items that always go together. Paying bills and adding money back into my checking account are things I always do together; to do otherwise risks disorder and insufficient funds.

I pulled details out of the "Home Depot" item and instead added an item further up the plan, reminding me to "Make a Home Depot list." If I was already going to put together a grocery list, I figured I might as well flip it over and put a hardware-store list on the other side. There's a lesson here: Plan-making is something that gets better with practice. Every time you draft a plan, you'll probably see some way of improving it.

On the other hand, sooner or later you have to stop fooling around and execute the plan.

I'm saying all this to get you in the habit of looking at a plan as something that works at different levels. A good plan can be reduced to an "at-a-glance" form that tells you the general shape of the things you want to do today, without confusing the issue with reams of details. On the other hand, to be complete, the plan *must* at some level contain all necessary details. Suppose I had sprained an ankle out in the garage and had to have someone else run errands in my place? In that case, the detailed plan would have been required, down to and perhaps including detailed driving directions to the various stores.

Had Carol been the one to take over errand-running for me, I might not have had to add a lot of detail to my list. When you live with a woman for forty-five years, you share a lot of context and assumptions with her. But had my long-lost cousin Tony from Wisconsin been charged with dashing around Phoenix Metro doing my day's errands, I would have had to write pages and pages to make sure he had enough information to do it right.

Or if my very distant relative Heinz Duntemann from Dusseldorf were visiting, I would have had to do all that, and write the plan in German as well.

Or…if my friend Sandron from Tau Ceti V (who neither knows nor cares what a wood deck is, and might consider Thompson's Water Seal a species of sports drink) volunteered to run errands for me, I would have had to explain even the minutest detail (over and above the English language and alphabet), including what stop lights mean, how to drive, our decimal currency (Sandron has sixteen fingers and counts in hex) and that cats are pets, not hors d'œrves.

To summarize: The shape of the plan—and the level of detail in the plan—depends on who's reading the plan, and who has to follow it. Remember this. We'll come back to it a little later.

1.2. C<small>OMPUTER</small> P<small>ROGRAMS AS</small> P<small>LANS OF</small> A<small>CTION</small>

If you're coming into this book completely green, the conclusion may not be obvious, so here it is: *A computer program is very much a "do-it" list for a computer, written by you.* The process of creating a computer program, in fact, is very similar conceptually to the process of creating a do-it list, as we'll discover over the course of Part 1.

Ted Nelson's description of a computer as a box that follows a plan is almost literally true. You load a computer program—the plan—into the computer's memory, and the computer will follow that plan, step-by-step, with a tireless persistance and absolute literal adherence to the letter of the plan.

I'm not going to go into a tremendous amount of detail here as to the electrical internals of computers. For that, you might pick up my book *Assembly Language Step By Step* (John Wiley & Sons, 2009; available on Amazon) and read its first several chapters, which explain the Intel CPU family and computer memory in simple terms. In 2014 I wrote the basic chapters in *Learning Computer Architecture with Raspberry Pi*, which explain the electrical reality of ARM-based computers, and go into much more detail. You can run FreePascal and Lazarus on the Raspberry Pi, and if you like Pascal (or don't feel like learning C or Python) I recommend giving them a try.

The notion of an "instruction set"

I had a very hard time catching onto the general concept of computing at first. Everybody who tried to explain it to me danced all around the issue of what a computer actually *was*. Yes, I knew you loaded a program into the computer and the program ran. I understood that one program could control the execution of other programs, and a host of other very high-level details. But I hungered to know what was underneath it all.

What I think was bothering me was the very important question: *How does the computer understand the steps that comprise the plan?*

The answer, like a plan, can be understood on several levels. At the highest level, you can think of it this way: The computer understands a very limited set of commands. These commands are the instructions that you write down, in order, when you sit at your desk and ponder the way to get the computer to do the things you want it to do. Taken together, the commands that the computer understands are called its *instruction set*. The instruction set is summarized in a book that describes the computer in detail. Programmers study the instruction set, and they write a program as a sequence of instructions chosen from that set.

Emily, the robot with a one-track mind

Let's consider a very simple thought-experiment describing a gadget that has actually been built (although many years ago) at a major American university. The gadget is, in fact, a crude sort of robot. Let's call the robot Emily. (The name is a tribute to a robotics project that appeared in *Popular Electronics* for March, 1962. I built Emily for my eighth grade science fair, and won an award.)

Picture Emily as a round metal can roughly the size and shape of a low footstool or a dishpan. Inside Emily are motors and batteries to power them, plus relays that switch the motors on and off, allowing the motors to run forward or backward, or to make right and left turns by running one motor or the other alone. On Emily's top surface are a slot into which a card can be inserted, and a button marked "GO."
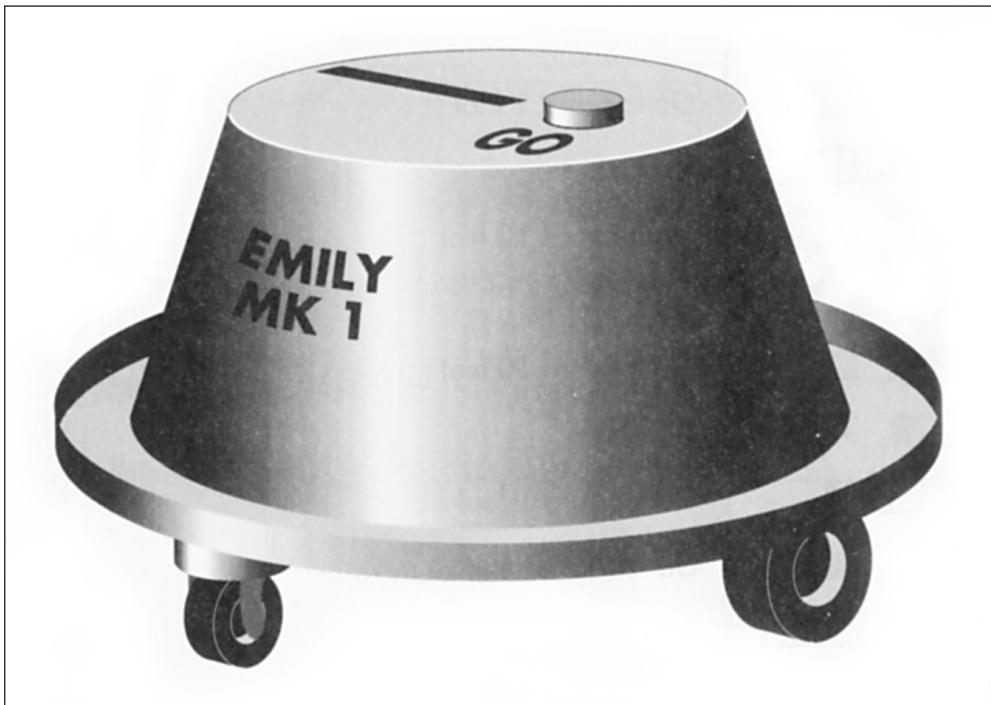
Figure 1.1. Emily the Robot

Left to her own devices, Emily does nothing but sit in one place. However, if you take a card full of instructions and insert the card into the slot on Emily's lid and press the "GO" button, Emily zips off on her own, stopping and going and turning and reversing. She's following the instructions on the card. Eventually she reaches and obeys the last instruction on the card, and simply stops where she is, waiting for another card and another press of the "GO" button.

Figure 1.2 shows one of Emily's instruction cards. The card contains thirteen rows of eight holes. In the figure, the black rectangles represent holes punched through the card, and the white rectangles are places where holes could be punched, but are not. Underneath the slot in Emily's lid is a mechanism for detecting holes by shining eight small beams of light at the card. Where the light goes through and strikes a photocell on the other side, there is a hole. Where the light is blocked, there is no hole.

The holes can be punched in numerous patterns. Some of the patterns "mean something" to Emily, in that they cause her machinery to react in a predictable way. When Emily's internal photocells detect the pattern that stands for "Go forward 1 foot" her motors come on and move her forward a distance of one foot, then stop. Similarly, when Emily detects the pattern that means "Turn right," she pivots on one motor, turning to the right. Patterns that don't correspond to some sort of action are ignored.

▮▯▯▯▯▮▯▮	Go forward 1 foot
▮▮▯▮▯▯▮▯	Turn left
▮▯▮▯▯▮▯▯	Go forward 10 feet
▮▮▯▮▯▯▮▯	Turn left
▮▯▯▯▯▮▯▮	Go forward 1 foot
▮▮▯▮▯▮▮▮	Turn right
▮▯▯▯▯▮▯▮	Go forward 1 foot
▮▯▯▯▯▮▯▮	Go forward 1 foot
▮▮▯▮▯▯▮▯	Turn left
▮▯▮▯▯▮▯▯	Go forward 10 feet
▮▮▯▮▯▯▮▯	Turn left
▮▮▯▮▯▮▯▯	Go forward 10 feet
▮▮▯▮▯▯▮▯	Turn left
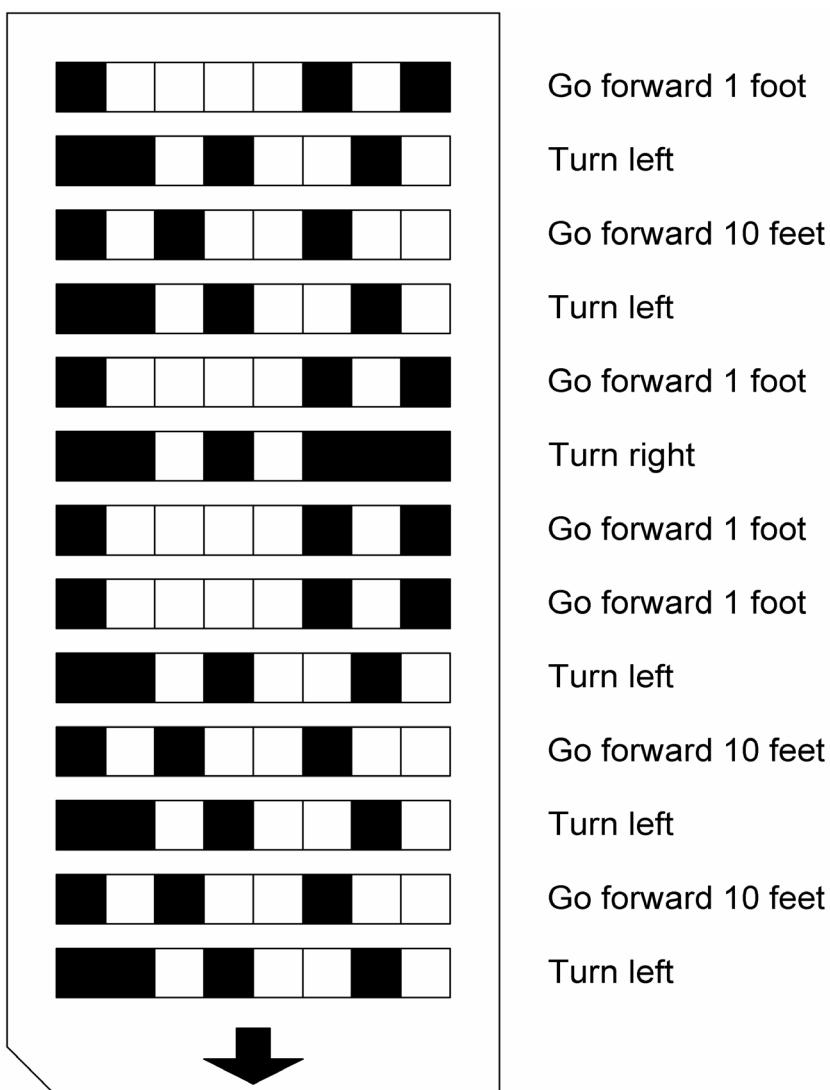
Figure 1.2. A program card for Emily the Robot

The card shown in Figure 1.2 describes a plan for Emily. It is literally a list of things that she must do, arranged in order from top to bottom. The arrow shows which end of the card must be inserted into Emily's card slot, with the cut corner on the left. Once the card is inserted into the slot in her lid, a press on her "GO" button sets her in motion, following the plan as surely as I followed mine by jumping into the 4Runner and heading off down Nevada Street to do my Saturday morning errands.

When Emily follows the plan outlined on the card, she moves in the pattern shown in Figure 1.3. It's not an especially sophisticated or useful plan—but then again, there's only room for thirteen instructions on the card. With a bigger card—or more slots to put cards into—Emily could execute much longer and more complex programs.
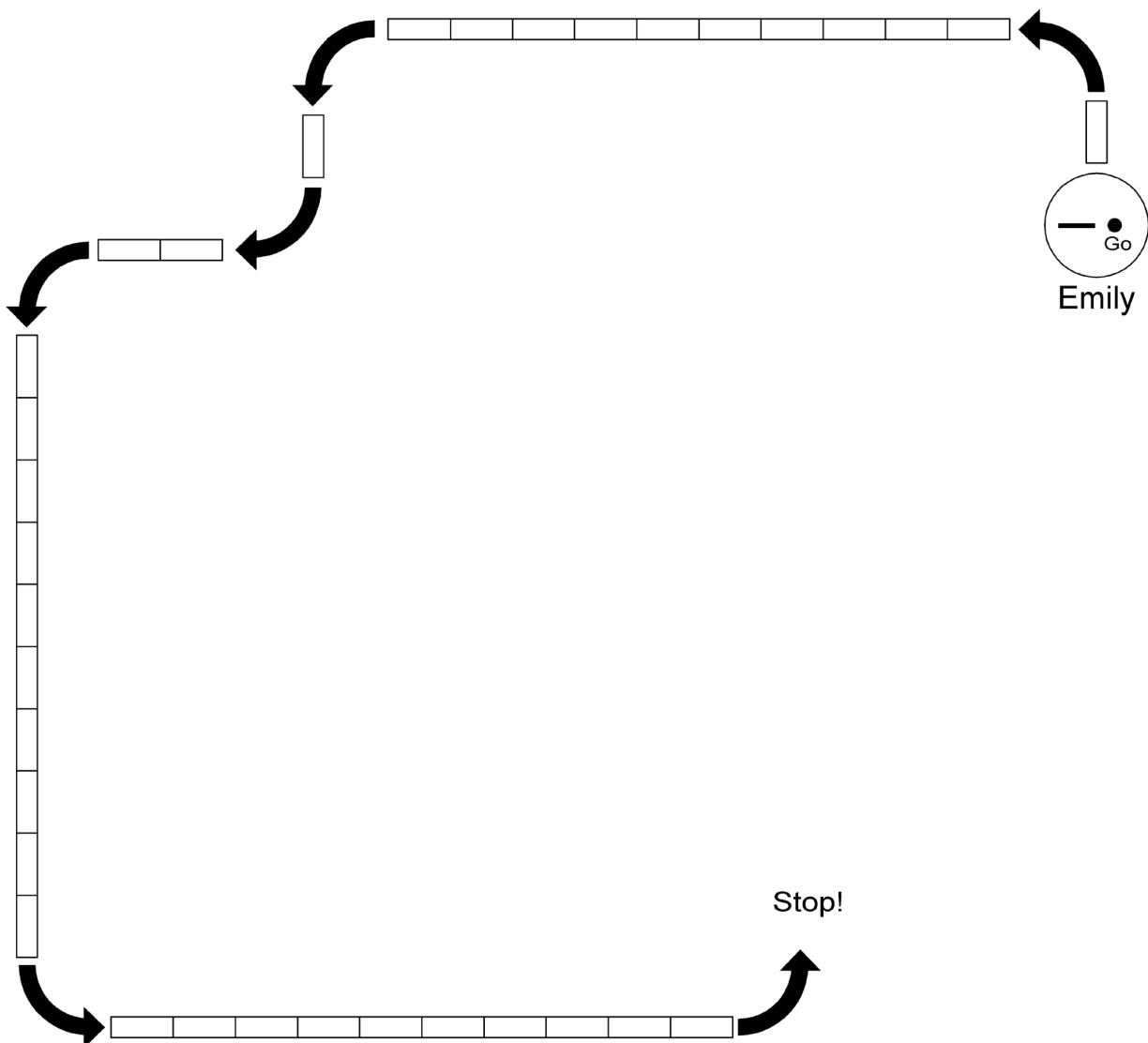
Figure 1.3. How Emily follows her instructions.

Emily's instruction set

It's interesting to look at the plan-card in Figure 1.2 and dope out how many different instructions are present on the card. The answer may surprise you: *four*. It looks more complex than that, somehow. But all that Emily is doing is executing sequences of the following instructions:

Go forward 1 foot
Go forward 10 feet
Turn left
Turn right

There's no instruction to stop; when Emily runs out of instructions, she stops automatically.

Now, Emily is a little more sophisticated than this one simple card might indicate. Over and above the four instructions shown above, Emily "understands" four more:

Go backward 1 foot
Go backward 10 feet
Rotate 180°
Sound buzzer

I've summarized Emily's full instruction set in Figure 1.4.

Go forward 1 foot

Go forward 10 feet

Turn right

Turn left

Go backward 1 foot

Go backward 10 feet

Rotate 180 degrees

Sound buzzer
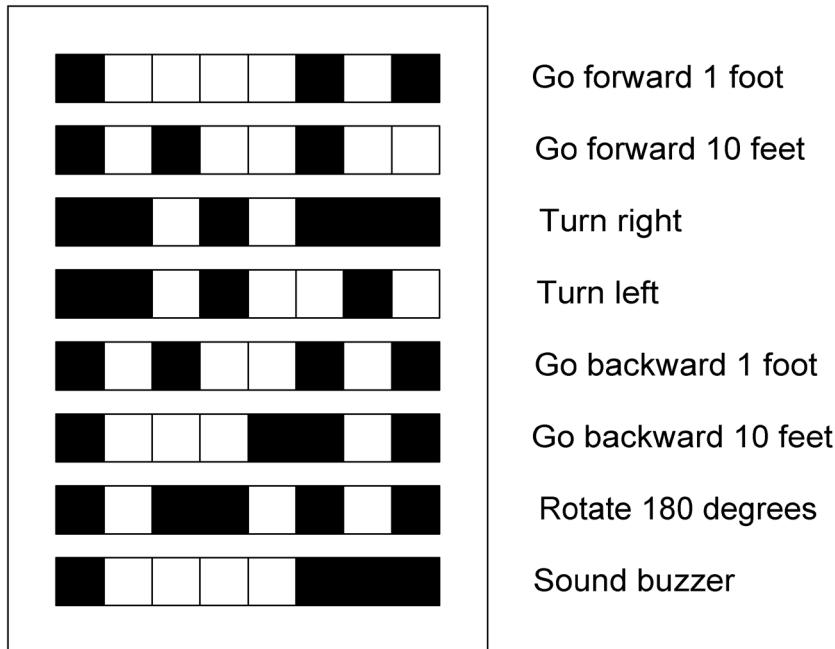
Figure 1.4. The Emily Mark I Instruction Set.

When you want to punch up a new card for Emily to follow, you must choose from among the eight instructions in Emily's instruction set. That's all she knows, and there's no way to make her do something that isn't part of the instruction set. However, it isn't always completely plain when something is or isn't part of the instruction set. Suppose you want Emily to move forward seven feet. There's no single instruction in Emily's instruction set called "Go forward seven feet." However, you could put seven of the "Go forward 1 foot" instructions in a row, and Emily

would go forward seven feet.

But that would take seven positions on the card, which only has thirteen positions altogether. Is there a way to make Emily move forward seven feet without taking so many instructions?

Consider the full instruction set, and then consider this sequence of instructions:

   Go forward 10 feet

   Go backward 1 foot

   Go backward 1 foot

   Go backward 1 foot

It's the long way around, in a sense, but the end result is indeed getting Emily to a position seven feet ahead of her starting point. It takes a little longer, timewise, but it only uses up four positions on the card.

This is a lot of what the skill of programming involves: Looking at the computer's instruction set and choosing sequences of instructions that get the job done. There is usually more than one way to do any job you could name—and sometimes an infinite number of ways, each with its own set of pluses and minuses. You will find, as you develop your skills as a programmer, that it's relatively easy to get a program to work—and a whole lot harder to get it to work *well*.

Different instructions sets

There is something I need to make clear: An instruction set is *not* the same thing as a program. A computer's instruction set is baked into the silicon of its CPU (Central Processing Unit) chip. (There have been computers—big ones—created with alterable instruction sets, but they are not the sorts of things we are ever likely to work on.) Once the chip is designed, the instruction set is almost literally set in stone.

However, as years pass, computer designers create new CPU chips with new instruction sets. The new instruction sets are sometimes massively different from the old ones, but in many cases, a new instruction set comes about simply by adding new instructions to an existing instruction set while designing a new CPU chip or family of CPU chips.

This was done when Intel designed the 80286 CPU chip in the early '80s. The dominant PC-compatible CPU chip up to that time was Intel's 8088, an 8-bit version of Intel's original 16-bit 8086. The 8088 was used by IBM in its original

PC and XT machines. Intel added a lot of computing muscle to the 8088 when it created the 80286, but the remarkable thing was that it only added capabilities— *Intel took nothing away.* The 80286's instruction set is larger than the 8088's, but it also *contains* the 8088's. That being the case, anything that runs on an 8088 also runs on an 80286. This has been the case as Intel's product line has evolved down the years through the 80386, 80486, the Pentium, and the more recent Core. All the instructions available in Intel's original 8086/8088 instruction set are still there in the latest Core CPUs. (Whether programs written for the 8088 will run on a Core i7 really depends more on the operating system than the CPU chip.)

Emily Mark II

We can return to Emily for a more concrete example. Once we've played with Emily for a few months, let's say we dismantle her and rebuild her with more complex circuitry to do different things. We add more sophisticated motor controls that allow Emily to make half-turns (45°) instead of just full 90° left and right turns. This alone allows tremendously more complex paths to be taken.

But the most intriguing addition to Emily is an electrically operated "tail" that can be raised or lowered under program control. Attached to this tail is a felt-tip reservoir brush, much like the ones sign painters use for large paper signs. One new instruction in Emily's enlarged instruction set lowers the brush so that it contacts the ground. Another instruction raises it off the ground so that it remains an inch or so in the air.

If we then put down large sheets of paper in the room where we test Emily, she can draw things on the paper by lowering the brush, moving around, and then raising the brush. Emily can draw a 1-foot square by executing the following set of instructions:

    Lower brush
    Go forward 1 foot
    Turn right
    Go forward 1 foot
    Turn right
    Go forward 1 foot
    Turn right
    Go forward 1 foot
    Raise brush

The full Emily Mark II instruction set is shown in Figure 1.5.

▮▯▯▯▯▮▯▮	Go forward 1 foot
▮▯▮▯▯▮▯▯	Go forward 10 feet
▮▮▯▮▯▮▮▮	Turn right
▮▮▯▮▯▯▮▯	Turn left
▮▯▮▯▯▮▯▮	Go backward 1 foot
▮▯▯▯▮▮▯▮	Go backward 10 feet
▮▯▮▯▯▮▯▮	Rotate 180 degrees
▮▯▯▯▯▯▮▮	Sound buzzer
▯▯▮▯▮▮▯▯	Turn half-left
▯▯▮▯▯▮▯▯	Turn half-right
▮▮▮▯▮▯▯▮	Raise brush
▮▮▮▯▯▯▯▮	Lower brush
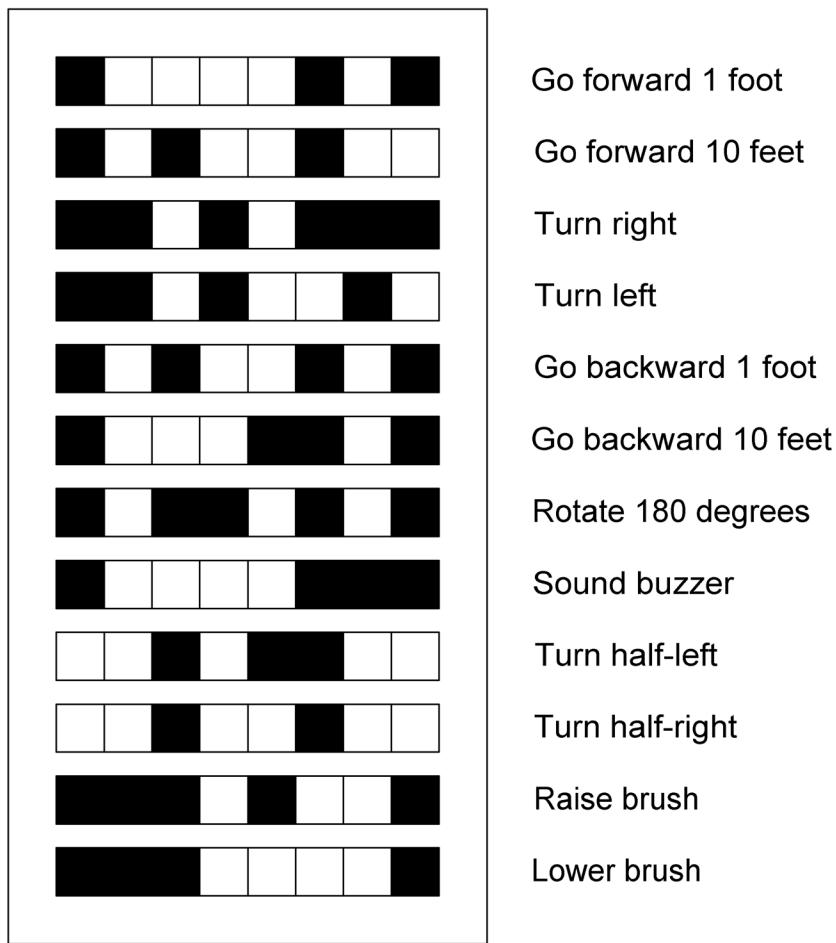
Figure 1.5. The Emily Mark II Instruction Set.

The whole point I'm making here is that a computer program is a plan for action, and the individual actions must be chosen from a limited set that the computer understands. If the computer doesn't understand a particular instruction, that instruction can't be used. Sometimes you can *emulate* a missing instruction by combining existing instructions to accomplish the same results. We did this earlier by using several instructions to make Emily act as though she had a "Move forward seven feet" instruction. Often, however, that is difficult or impossible. How, for example, might you make Emily perform a half-left turn by combining sequences of her original eight instructions? Easy. *You can't.*

This is one way to understand the limitations of computers. A computer has a fundamental instruction set. This instruction set is fairly limited, and the individual instructions are very tiny in their impact. An instruction, for example, might simply add 1 to a location in memory. Through a great deal of cleverness, this elemental instruction set can be expanded enormously by combining a multitude of tiny, limited instructions into larger, more complex instructions. One such mechanism is the subject of this book: FreePascal, as I'll explain in the next chapter.

1.3. Changing course inside the plan

Back in the DOS era (basically 1981-1995) if you used a PC for any amount of time, you probably wrote small "batch" files to execute sequences of DOS commands or utility programs. Virtually everyone back then periodically tinkered with AUTOEXEC. BAT, which was the batch file in charge of setting up your machine at boot-time by loading resident programs, changing video modes, checking remaining hard disk space, and things like that.

A DOS batch file was very much what we've been talking about: a "do-it" list for your computer. I had a whole lot of them, most created to take the three or four small steps necessary to invoke some application. When I wanted to work with my address book file using the Paradox database application, I used to use this little batch file:

```
D:
CD \PARADOX3\PERSONAL
VHR MDS /N
PARADOX3
```

It's not much, but it saved me having to type these four lines every time I wanted to update someone's phone number in my address book database. (We forget sometimes what tremendous time savers Windows and Mac OS have been by handling things like that for us, and retaining applications in memory for quick use at any time.) The first command shown above moves me to the D: hard disk drive. The second command moves me into the subdirectory containing my address book database. The third command invokes a special utility program that resets and clears the unusual monitor and video board that I used for years in the DOS era. The fourth and last line invokes the Paradox 3.0 database program.

Yes, DOS batch files are ancient and little-used these days, but they perfectly illustrate my point: Like most people's "do-it" lists, DOS batch files run straight through from top to bottom, in one path only.

Departing from the straight and narrow

Well, how else would it run? Well, a batch program (for they truly were computer programs) might contain a loop or a branch, which alters the course of the plan based on what happens while the plan is underway. (Branching was possible with DOS batch, but only deep geeks ever did much of it.)

You do that sort of thing all the time in daily life, mostly without thinking. Suppose, for example, you go grocery shopping with the item "poppy-seed rolls"

on your grocery list. Now when you get to Safeway, suppose it's late in the day and the poppy-seed rolls are long gone. You're faced with a decision: What to buy? If you're having hamburgers for supper, you need something to put them on. So you buy Mother Ersatz' Genuine Bread-Flavored Imitation Hamburger Buns. You didn't write it down this way and may not even think of it this way, but your "do-it" list contains this assumed entry:

>   If the bakery poppy-seed rolls are gone,
>   buy Mother Ersatz' Buns.

Then again, if Mother Ersatz' products make you feel, well...ersatz, you might change your whole supper strategy, leave the frozen hamburg in the freezer, and gather materials for stir-fry chicken instead:

>   If the bakery poppy-seed rolls are gone, then do this:
>   Buy package of boneless chicken breasts;
>   Buy bottle of teriyaki sauce
>   Buy fresh mushrooms
>   Buy can of water chestnuts

Most of the time we perform such last-minute decision-making "on the fly," hence we rarely write such detailed decisions down in our original and carefully crafted "do-it" lists. Most of the time, Safeway *will* have the poppy-seed rolls and we operate under the assumption that they will always be there. We think of "Plan B" decisions as things that happens only when the world goes ballistic on us and acts in an unexpected fashion.

In computer programming, such decisions happen all the time, and programming would be impossible without them. In a computer program, little or nothing can be assumed. There's no "usual situation" to rely on. Everything has to be explained in full. It's rather like the situation that would occur if I sent Sandron the alien out shopping for me. I'd have to spell the full decision out in great detail:

>   If the store bakery poppy-seed rolls are still available,
>   then buy one package,
>   otherwise buy one package of Mother Ersatz' Buns.

This is an absolutely fundamental programming concept that we call the **IF.. THEN..ELSE** branch statement. (The word "otherwise" stands in well for the Pascal term **ELSE**.) It's a fork in the plan; a choice of two paths based on the state of things as they exist at that point in the plan. You can't *know* when you head out the door on your way to Safeway whether everything you want will be there...so you go in prepared to make decisions based on what you find when you get to the bakery department. In Pascal programming you'll come to use branches like this so often it will almost be done without thinking.

Doing part of the plan more than once

Changing the course of a plan doesn't necessarily mean branching off on a whole new trajectory. It can also mean going back a few steps and repeating something that may need to be done more than once.

A simple example of a loop to get a job done comes up often in something as simple as a recipe. When you're making a cake from scratch and the recipe book calls for four cups of flour, what do you do? You take the 1-cup measuring cup, dip it into the flour canister, fill it brim-full, and then dump the flour it contains into the big glass bowl.

Then you do exactly the same thing a second time...

...and a third time...

...and a fourth time.

The plan (here, a cake recipe) calls for flour. Measuring flour is done in one-cup increments, because you don't have a four-cup measuring cup. It's a little like the situation with Emily the Robot, when she has to move three feet forward. The instruction set doesn't allow you to measure out four cups of flour in one swoop, so you have to fake it by measuring out one cup four times.

In the larger view, you've entered a *loop* in the plan. You go through the component motions of measuring out a cup of flour. Then, you ratchet back just far enough in the plan to go through the same motions again. You do this as often as you must to correctly accomplish the recipe.

Counting passes through the loop

You've probably been in the situation where you begin daydreaming after the second cup, and by the time you shake the clutter out of your head, you can't remember how many times you've already run through the loop, and either go one too many or one too few. Counting helps, and being as how I'm a champion daydreamer, I'm not afraid to admit that when the count goes more than three or four, I start making tick

marks on a piece of scratch paper for each however much of whatever I throw into the bowl. This makes for much better cakes. (Or at least more predictable ones.)

You might write out (for cousin Heinz or perhaps Sandron the alien) this part of the plan like so:

> Do this exactly four times (and count them!):
>> Dip the measuring cup into the flour canister;
>> Fill the cup to the line with flour;
>> Dump the flour into the mixing bowl.

This is another element that you'll see very frequently in Pascal programming. It's called a **FOR..DO** loop, and we'll return to it later in this book.

Doing part of the plan until it's finished

There are other circumstances when you need to repeat a portion of a plan until... well, until what must be done is done. You do it as often as necessary.

It's something like the way Mary Jo Mankiewicz measures out jelly beans at Candy'N'Stuff. You ask her for a quarter-pound of the broccoli-flavored ones. She takes the big scoop, holds her nose, and plunges it deep into that rich green bin. With a scoop full of jelly beans, she stands over the scale, and repeatedly shakes a dribble of jelly beans into the scale's measuring bowl until the digital display reads a little more than .25.

Written out for Sandron the Alien, the plan might read this way:

> Dig the scoop into the jelly beans and fill it.
> Take the full scoop over to the digital scale.
> Repeat the following:
>> Shake a few jelly beans from the scoop into the scale's bowl;
> Until the scale's digital readout reads 0.25 pounds.

Exactly how many times Mary Jo has to shake the scoop over the scale depends on what kind of a shaker she is. If she's new on the job and cautious because she doesn't want to go too far, she'll just shake one or two jelly beans at a time onto the scale. Once she wises up, she'll realize that going over the requested weight doesn't matter so much, and she'll shake a little harder so that more jelly beans drop out of the scoop with each shake. This way, she'll shake a lot fewer times, and when she

ends up handing you half a pound of jelly beans, that's OK—since one can't ever have enough broccoli-flavored jelly beans, now, can one?

Computers, of course, *do* mind when they go over the established weight limit, and they don't mind making small shakes if that's what it takes to get the final weight bang-on. A computer, in fact, is a tireless creature, and would probably shake out only one bean at a time, testing the total weight after each bean falls into the bowl.

This sort of loop is called a **REPEAT..UNTIL** loop, and it's also common in Pascal programming, as I'll demonstrate later on in this book.

The shape of the plan

The whole point I'm trying to make in this section is that a plan doesn't have to be shaped like one straight line. It can fork, once or many times. Portions of the plan can be repeated, either a set number of times, or as many times as it takes to get the job done. This is nothing new to any of us—we do this sort of thing every day in muddling through our somewhat overstuffed lives. In fact, if you've taken any effort at all to live an organized life, you'll probably make a dynamite programmer, since success in life or in programming cooks down to creating a reasonable plan and then seeing it through from start to finish without making any (serious) mistakes.

1.4 Information and Action

Actually, we've only talked about half of what a plan (in computer terms) actually is. The other half, which some people say is by far the more important half, has been waiting in the wings for its place in this discussion.

That other half is the stuff that is acted upon when the computer works through the plan you devise for it. When we spoke of Mary Jo Mankeiwicz measuring out jelly beans, we focused on the way she ladled them out. Just as important to the process are the jelly beans themselves. And so it is, whether you characterize the plan as shopping for groceries, measuring out flour for a recipe, or building a birdhouse. The shopping, the measuring, and the building are crucial—but they mean nothing without the groceries, the flour, and those little pieces of plywood that always split when you try to get a nail through them.

In computer terms, the stuff that a program acts on is information, by which I mean symbols that have some meaning in a human context.

Code vs. data

When you create a computer program, the plan portion of the program is a series of steps, written in a programming language of some sort. In this book, that's going to mean FreePascal, but there are plenty of programming languages in the world (too many by half—or maybe three quarters, I think) and they all work pretty much the same way. Taken together, these steps are called *program code,* or simply code. Collectively, the information acted on by the program code is called *data*. The program as a whole contains both code and data.

My friend Tom Swan (who has written some terrific books on the Pascal programming language himself) says that code is what a computer program *does,* and data is what a computer program *knows*. This is a very elegant way of characterizing the difference between program code and data, and I hesitate in using it mostly because too many people have swallowed the Hollywood notion of mysteriously intelligent computers a little too fully. A program doesn't "know" anything—it's not alive, and a consensus is beginning to form that computers can never truly be made to think in the sense that human beings think. Roger Penrose has written a truly excellent book on the subject, *The Emperor's New Mind,* which is difficult reading in places but really nails the whole notion of "artificial intelligence" to the wall. The subject is a big one, and an important one, and I recommend the book powerfully.
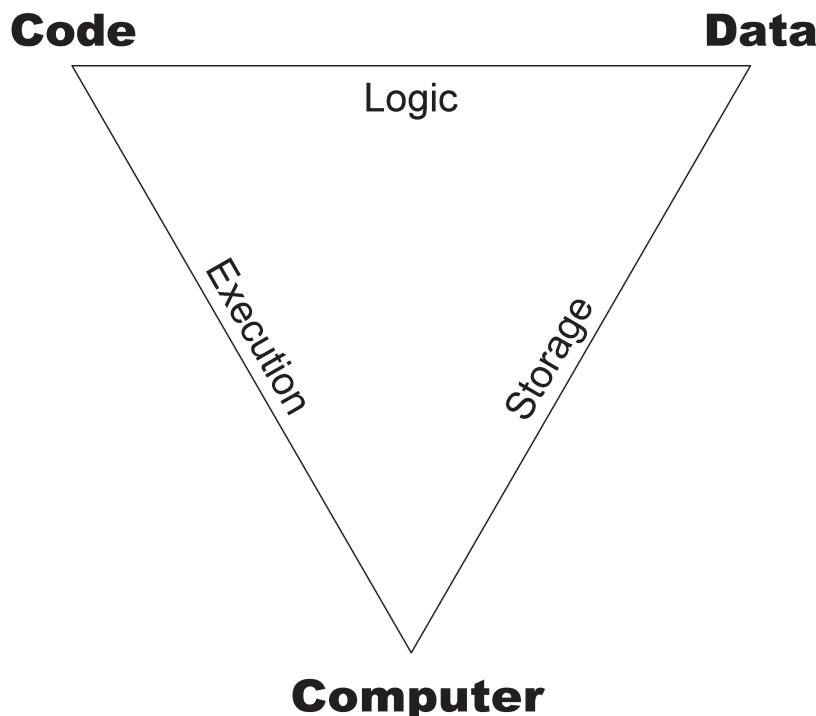

Figure 1.6. Computer, Code, and Data.

But there's another reason: Thinking of code and data as "doing" and "knowing" leaves out an essential component: The gizmo that does the doing and the knowing; that is, the computer itself. I recommend that you always keep the computer in the picture, and think of the computer, code, and data as an unbreakable love triangle. No single corner is worth a damn without both of the other two. Between any two corners you'll read what those two corners, working together, create. See Figure 1.6. In summary: The program code is a series of steps. The computer takes these steps, and in doing so manipulates the program's data.

Let's talk a little about the nature of data.

Let X be...

A lot of people shy away from computer programming due to math anxiety. Drop an "X" in front of them and they panic. In truth, there's very little math involved in most programming, and what math there is very rarely goes beyond the high-school level. In virtually all programs you're likely to write for data processing purposes (as opposed to scientific or engineering work) you'll be doing nothing more complex than adding, subtracting, multiplying, or (maybe) dividing—and *very* rarely taking square roots.

What programming *does* involve is symbolic thinking. A program is a series of steps that acts upon a group of symbols. These symbols represent some reality existing apart from the computer program itself. It's a little like explaining a point in astronomy by saying, "Let this tennis ball represent the Earth, and this marble represent the Moon..." Using the tennis ball and the marble as symbols, you can show someone how the moon moves around the earth. Add a third symbol (a soccer ball, perhaps) to represent the Sun, and you can explain solar eclipses and the phases of the Moon. The tennis ball, marble, and soccer ball are placeholders for their real-life rock-and-hot-gas counterparts. They allow us to think about the Earth, Sun, and Moon without getting boggled by the massiveness of scale on which the solar system operates. They're *symbols*.

A *bucket, a label, and a mask*

Using a couple of balls to represent bodies in the solar system is a good example of an interesting process called *modeling*, which is a common thing to do with computer programs, and we'll come back to it at intervals throughout this book. But data is simpler even than that. A good conceptual start for thinking about data is to imagine a bucket.

Actually, imagine a group of buckets. (Your programs won't be especially useful if they only contain one item of data.) Since the buckets all look pretty much alike,

you'd better also imagine that you can slap a label on each bucket and write a name on each label.

The buckets start out empty. You can, at will, put things in the buckets, or take things out again. You might place ten marbles in a bucket labeled "Ralph," or removed five marbles from a bucket labeled "George." You can look at the buckets and compare them: Bucket Ralph now contains the same number of marbles as bucket George—but bucket Sara contains more marbles than either of them. This is pretty simple, and maps perfectly onto the programming notion of defining a *variable*—which is nothing more than a bucket for data—and giving that variable a name. The variable has a *value*, meaning only that it contains something. The subtlety comes in when you build a set of assumptions about what data in a variable *means*.

You might, for example, create a system of assumptions by which you'll indicate truth or falsehood. You make the following assumption: A bucket with one marble in it represents the logical value True, whereas an empty bucket (that is, one with no marbles in it) represents the logical value False. Note that this does *not* mean you write the word "True" on one bucket's label or "False" on another's. You're simply treating the buckets as symbols of a pair of more abstract ideas.

With the above assumptions in mind, suppose you take a bucket and write "**Mike Is At Home**" on its label. Then you call Mike on the phone. If Mike picks up the phone, you drop one marble into the labeled bucket. If Mike's phone simply rings (or if his answering machine message greets you) you leave the labeled bucket empty.

With the bucket labeled "**Mike Is At Home**" you've stored a little bit of information. If you or someone else understands the fact that one marble in a bucket indicates True, and an empty bucket indicates False, then the bucket labeled "Mike Is At Home" can tell you something useful: That it is *true* that Mike is at home. Just look in the bucket—and save yourself a phone call.

What's important here is that we've put a sort of mask on that bucket. It's not simply a bucket for holding marbles anymore. It's a way of representing a simple fact in the real world that may have nothing whatsoever to do with either marbles or buckets.

Much of the real "thinking" work of programming consists of devising these sets of assumptions that govern the use of a group of symbols. You'll be thinking things like this:

> Let's create a variable called **BeanCount**. It will represent the number of jelly beans left in the jelly bean jar. Let's also create a

> variable called **ScoopCapacity**, which will represent the number
> of jelly beans that the standard scoop contains when filled.

In both instances, the variable itself is simply a place somewhere inside the computer where you can store a number. What's important is the mask of meaning that you place in front of that variable. The variable contains a number—*and that number represents something*. This is the fundamental idea that underlies all computer data items. They're all buckets with names—for which you provide a meaning.

1.5. The Craftsman, His Pockets, His Workbench, and His Shelves

Let's turn our attention again to Ted Nelson's aphorism that *a computer is a box that follows a plan.* So far I've spoken about the plan itself—how it runs from top to bottom, perhaps branching or looping along the way. I've talked a little bit about data—the stuff that the program works on when it does its work. However, only half of the essence of computing is the plan and its data, and the other half is the nature of the box that follows the plan.

It's time to talk about the box.

Divided into three parts

Your PC (and virtually all other kinds of computers that have ever been designed) is divided into three general areas: The *Central Processing Unit* (CPU), *storage,* and *input/output* (I/O).

The CPU is the boss of the machine. It's the part of the machine that contains the instruction set we spoke of earlier. The instruction set, if you recall, is that fundamental list of abilities that the computer has. All programs, no matter how big or how small, and regardless of how simple or how complex, are composed of some sequence of those fundamental instructions.

Storage is another term for memory, although it actually includes what we call Random Access Memory (RAM) and disk storage as well. Storage is where program code and data lives most of the time. RAM is storage that exists right next to and around the CPU, on silicon chips. The CPU can get into RAM very quickly. The disadvantage to RAM is that it's expensive, and, (worse) it goes blank when you turn power to the computer off. Disk storage, by contrast, is "farther away" from the CPU and is much slower to read from and write to. Balancing that disadvantage is the fact that disk storage is cheap and (better still) once something is written onto a disk, it stays there until you erase it or write something over it.

Input and output are similar, and differ mainly in the direction that information moves. Your keyboard is an input device, in that it sends data to the CPU when you press a key. Your screen is an output device, in that it puts up on display information sent to it by the CPU. The parallel port to which you connect your printer is another output device, and your Ethernet network port swings both ways: It both sends and receives data, and thus is both an input and an output device at once.

What happens when a program runs

Programs are stored on disk for the long haul. When you execute a program, the CPU brings the program from disk storage into RAM. That done, the CPU begins at the top of the program in RAM and begins "fetching" instructions from the program in RAM into itself. One by one, it fetches instructions from RAM and then executes them. It's a process much like reading a step from a recipe and then performing that step. You first need to know (by reading the receipe) that you must throw four cups of flour into the bowl, and then you have to go ahead and actually measure out the flour and put it into the bowl.

During a program's execution, the CPU may display information on the screen or ask for information from the keyboard. There's nothing special about input or output like this; there are instructions in the CPU's instruction set that take care of moving numbers and characters in from the keyboard or out to the screen.

What *is* more intriguing is the notion that the plan—that is, the sequence of instructions being executed by the CPU—may change based on data that you type at the keyboard. There are forks in the road (usually a multitude of them) and which road the CPU actually follows depends on what you type into data entry fields or answer to questions the CPU asks you.

You may see a question displayed on the screen something like this:

```
Do you want to exit the program now? (Y/N:)
```

If you press the "Y" key, one road at the fork will be taken, and that fork leads to the end of the program. If you press the "N" key, the other road at the fork will be taken, back into the program to do some more work.

Programs like this are said to be *interactive*, since there's a constant conversation going on between the program and you, the user. Inside the computer, the program is threading its way among a great many possible paths, choosing a path at each fork in the road depending on questions it asks you, or else depending on the results of the work it is doing.

Sooner or later, the program either finishes its work or is told by you to pack up for the day, and it "exits," which means that it returns control to the operating system, whatever the operating system might be.

Inside the box

That's how things appear to you, outside the box, sitting at the keyboard supervising. Understanding in detail what happens inside the box is the labor of a lifetime (or at least often seems to be) and is the subject of this entire book—and hundreds of others, including all the other books that I've ever written.

But a good place to start is with a simple metaphor. The CPU is a craftsman, trained in a set of skills that involves the manipulation of data. Just as a skilled machinist manipulates metal, and a carpenter manipulates wood, the CPU manipulates numbers, characters, and symbols. We can think of the CPU's skills as those instructions in its instruction set. The carpenter knows how to plane a board smooth—and the CPU knows how to add two numbers together to produce a sum. The carpenter knows how to nail two boards together—and the CPU knows how to compare two numbers to see if they are equal.

The CPU has a workbench in our metaphor: The computer's RAM. RAM is memory inside the computer, close to the CPU and easily accessible to the CPU. The CPU can reach anywhere into RAM it needs to at any time. It can choose a place in RAM "at random," and this is why we call it Random Access Memory. As the CPU stands in front of its workbench, it can reach anywhere on the workbench and grab anything there. It can move things around on the workbench. It can pick up two parts, put them together, and then place the joined parts back down on the workbench before picking up something else.

But before beginning an entirely different project, the CPU, being a good craftsman, will tidy up its workbench, put its tools away, and sweep the shavings into the trashbin, leaving a clean workbench for the next project.

Disk storage is a little different. In our metaphor, you should think of disk storage as the set of shelves in the far corner of the workshop, where the craftsman keeps tools, raw materials, and incomplete projects-in-progress. It takes a few steps to go over to the shelves, and the craftsman may have to get up on a stepstool to reach something on the highest shelves. To avoid running itself ragged (and wasting a lot of time) going back and forth to the shelves constantly, the CPU tries to take as much as it can from the shelves to its workbench, and stay and work at the workbench.

It would be like buying a birdhouse kit, opening the package, leaving the opened package on the shelves, and then traipsing over the to shelves to pull each piece out of the birdhouse kit as you need it while assembling the birdhouse.

That's dumb. The CPU would instead take the entire kit to the workbench and assemble it there, only going back to the shelves for something too big to fit on the workbench, or to find something it hadn't anticipated needing.

One final note on our metaphor: The CPU has a number of storage locations that are actually inside itself, closer even than RAM. These are called *registers,* and they are like the pockets in a craftsmen's apron. Rather than placing something back on the workbench, the craftsman can tuck a small tool or part conveniently in a pocket for very quick retrieval a moment later. The CPU has a few such pockets, and can use them to tuck away a number or a character for a moment.

So there are actually three different types of storage in a computer: Disk storage, RAM, and registers. Disk storage is very large (these days, trillions of bytes is not uncommon in hard drives) and quite cheap—but the CPU has to spend considerable time and effort getting at it. RAM is much closer to the CPU, but it is a lot more expensive and rarely anything near as large. Few computers have more than 4 or 8 gigabytes of RAM these days. Finally, registers are the closest storage to the CPU, and are in fact *inside* the CPU—but there are only a half dozen or so of them, and you can't just buy more and plug them in.

Summary: Truth or metaphor?

The electrical reality of a computer is complicated in the extreme, and getting moreso all the time. I've discussed it to a much greater depth in my books *Assembly Language Step By Step* and *Learning Computer Architecture with Raspberry Pi* than I can possibly discuss it here, and if you're curious about what RAM "really" is, those would be good books to read.

I'm not going to that depth in *this* book because FreePascal shields you from having to know as much about the computer's electrical structure and deepest darkest corners. I'm sticking with metaphor because a computer program *is* a metaphor. A program is a symbolic metaphor for a frighteningly obscure torrent of electrical switching activity ultimately occurring in something about the size of your thumbnail.

When we declare a variable in FreePascal (as I'll be explaining shortly) we're doing nothing different from saying, "Let's say that these eight transistor storage cells represent a letter of the alphabet, and we'll give it the name **DiskUnit**."

FreePascal, in fact, is a tool entirely devoted to the creation and perfection of such metaphors. You could create a simple program that modeled a shopping trip taken by Sandron the alien to Safeway, just as I described earlier in this chapter. There was, in fact, an intriguing software product available years ago called "Karel the Robot" which was a simulation of a robot on the computer screen. Karel could be given

commands and would follow them obediently, and the net effect was a very good education on the nature of computers.

This chapter has been groundwork, basically, for people who have had absolutely no experience in programming. All I've wanted to do is present some of the fundamental ideas of both computing and programming, so that we can now dive in and get on with the process of creating our program metaphors in earnest.

CHAPTER 2.
THE NATURE OF
SOFTWARE DEVELOPMENT

As I hinted in the last chapter, a computer program is a plan, written by you, that the computer follows in order to get something done. The plan consists of a series of steps that must be taken, and some number of decisions to be made along the way when a fork in the road turns up.

That's programming in the abstract, as simply put as possible. There are a lot of ways of actually writing a program, each way focusing on a different *programming language*. In this book, I'll be talking about only one programming language, the one called Pascal. Furthermore, I'll be focusing on only one single "dialect" of that language, FreePascal. FreePascal is itself very similar to the venerable Turbo Pascal from Borland, which has been with us since 1983. Nearly all program code written for Turbo Pascal (including Turbo Pascal's big brother Borland Pascal) will compile and run correctly under FreePascal.

This is not a limitation. Borland's Pascal implementations pretty much plowed all other Pascal dialects under the soil by 1990. If you're going to learn Pascal programming at all, you might as well learn the dialect comprising probably 95% of all Pascal compilers ever sold.

Look no further. You won't find anything better.

2.1. LANGUAGES AND DIALECTS

In *Star Trek IV: The Voyage Home,* the crew of the *Enterprise* travel back in time to 1987 San Francisco. When Engineer Scotty needs to use a computer, he is shown a Macintosh and immediately picks up the mouse as though it were a microphone and begins addressing the poor machine directly:

"Computer: We're going to design a molecular structure for transparent aluminum!"

The Mac had very little to say in reply. Computers didn't understand the English language very well back in 1987, and they don't understand it much better today. So what *do* computers understand? What is their native language?

Assembly language

The fast answer is that computers understand something called "assembly language," in which each step in the plan is one of those fundamental machine instructions I described conceptually in Chapter 1. Machine instructions are incredibly minute in what they do; for example, a single instruction may do nothing more than fetch a byte of data from a location in RAM and store it in a location inside the CPU called register AX. It takes an *enormous* number of such instructions to do anything useful; hundreds or thousands for small programs, and many hundreds of thousands or even millions for major application programs like Microsoft Excel or AutoCAD.

The instructions themselves are terse, cryptic, and look more like something copied out of a mad scientist's notebook than anything you or I would call a language:

```
MOV     EAX,[EBX]
SUB     EAX,EDX
AND     EAX,0FF0DH
DEC     ECX
LOOPNZ  MSK13
```

Actually, the frightening truth is that even these cryptic statements are themselves "masks" for the *true* machine instructions, which are nothing more than sequences of 0's and 1's:

```
0110101000110010
0000000101110011
1110111110011011
0110110110001010
```

It's possible to program computers by writing down sequences of 0's and 1's and somehow cramming them into a computer through toggle switches, with an "up" switch for a "1" and a "down" switch for a "0". I used to do this in 1976 (for my home-made computer called the COSMAC ELF) and thought it was great good fun, because back then it was the best that my machine and I could do.

Is it really fun?

Mmmmm...no. In truth, it gets old in a *big* hurry. Doing something as simple as making the PC's speaker beep requires fifteen or twenty machine instructions, all laid out *precisely* the right way. Even writing assembly language in almost-words like **MOV EAX,[EBX]** is tiresome and done today only by the most curious and the most dedicated among us. But the computer only understands machine instructions.

What to do?

High-level computer languages

The answer dates back almost to the dawn of computing. (As I said, writing and debugging machine instructions gets old in a big, *big* hurry...) Early on, people defined what we call *high-level computer languages* to do much of that meticulous machine-instruction arranging automatically.

In a high-level language, we define words and phrases that mean something to us and perform some simple task on the computer. A good example is beeping the PC's speaker. We might decide that the word **BEEP** will be used to indicate that the computer's speaker is to be sounded. That done, we write down the sequence of machine instructions that actually causes sound to be generated on the speaker, and we associate those machine instructions with the recognizable term **BEEP**.

This sequence of instructions remains consistent and never changes. So we create a program for ourselves that, when it sees the word **BEEP**, substitutes the sequence of twenty machine instructions that actually does the speaker-beeping. We only need to remember that the word **BEEP** does the beeping. Our clever program, called a *compiler*, remembers the twenty-instruction sequence that accomplishes the beep, so that we don't have to. (Perhaps our program isn't especially clever. But it remembers things *very* well...)

We go on from there and define other easily-readable words and phrases that stand for sequences of dozens or even hundreds of machine instructions. This allows us to write down easily readable commands like the following in only a few seconds:

```
Remainder := Remainder - 1;
IF Remainder = 0 THEN
  BEGIN
    BEEP;
    Write('Warning!  Your time has run out!');
  END;
```

Once you have a feeling for the high-level language, you can look at sequences like this and know exactly what they'll do without stretching your brain too much. The reality is that it may take hundreds of machine instructions for the CPU to do the work involved in what we have written, but for *our* eyes, it's only a few short lines.

Programs that write other programs

This clever compiler program understands a great many English-like words and phrases. We create a disk file, like a word processor file, containing sequences of English-like words and phrases. The compiler program reads in the file of English-

like words and phrases, and writes out an equivalent file of machine instructions. This file of machine instructions can be loaded and executed by the CPU. But even though this program file consists of thousands or tens of thousands of machine instructions, *we never had to know even a single machine instruction to write it*. All we had to do was understand how the English-like commands of the high-level language affect the machine. The compiler takes care of the "ugly" stuff like remembering which sequence of twenty machine instructions beeps the speaker. All we have to remember is what **BEEP** does.

Much better!

A compiler program is thus a program that writes other programs, with some direction from us. It does its job so well that we can actually forget all about what happens with the machine instructions (most of the time, anyway) and concentrate on the logic of how the English-like words and phrases go together.

Different languages

A host of high-level languages exists for the PC. Pascal, C, C++, C#, Python, Javascript, and BASIC are the most common, but there are hundreds of others with obscure or puckish names like COBOL, Perl, Java, Forth, APL, Smalltalk, Eiffel, PHP, PL/1, Rexx, FORTRAN, Lisp, Scheme, and on and on and on. As different as they may seem on the surface, they all do the same thing underneath: Arrange machine instructions to accomplish the work that we encode in English-like words and phrases.

Pascal, for example, uses the word **Write** to display information on the screen. BASIC, by contrast, uses the word **PRINT**. The C and C++ languages use the word **printf**. Forth uses the word **TYPE**. Others use words like **SAY**, **OUTPUT**, or **Show**. Those words were chosen by the people who designed each language for reasons they considered good ones. However, what those different words *do* is all pretty much the same.

A high-level language is defined as a set of commands, plus a set of rules as to how these commands are used alone and combined. These rules are called the *syntax* of the language, and they correspond roughly to the syntax and grammar of spoken human languages like English and German. Computer languages are not as rich in expression as human languages, but they are *much* more precise—and needless to say, they "speak" only of things that a computer can actually accomplish.

Dialects of a single language

Having a mob of hundreds of dissimilar languages to choose from might seem confusing enough. Unfortunately, even within a single language, there are variations

on a theme called *dialects*. Each person or company who creates a compiler that understands a given computer language might construct the compiler to understand things a little differently from other compilers written earlier for the same language. Thus not all Pascal compilers agree on what certain program commands mean. Nor do all BASIC compilers agree on the syntax and command set of BASIC. If you write a program in Drs. Kemeny & Kurtz's TrueBasic it won't necessarily compile correctly if you hand that same program to Microsoft's Visual BASIC.

Dialects usually happen when companies who write compilers add new features and abilities to languages, in order to produce a more powerful language or (at least) one perceived as different from the compilers already on the market.

FreePascal is a dialect of the Pascal language. Pascal has been around for almost fifty years now (since 1971) and it's done some serious growing in the process. Pascal predates PCs; in fact, it predates all microcomputers of any design and was originally created to run on massive mainframe computers, those famous for being kept behind locked doors in air-conditioned rooms with raised floors. The man who designed Pascal, Niklaus Wirth, was trying to prove a point in computer science when he designed Pascal, and really didn't intend to write an exhaustive and generally useful language. Other companies added features to Pascal over the years, and little by little the language broke into mutually-exclusive dialects that were about 90% common. Alas, in computer languages as in horseshoes, "almost" just doesn't count.

For example, Wirth's original Pascal wrote information to disk files with the **Put** command. Borland's Turbo Pascal broke with this concept in the early 1980s by using the **Write** command to write data to disk, and omitted the **Put** command completely. FreePascal does things the Turbo/Borland Pascal way. If you take a program written in the original version of Pascal and try to compile it using FreePascal, the compiler will display an error message if it encounters a **Put** command, since it doesn't know what sequence of machine instructions **Put** is supposed to represent.

This problem of dialects is worse in most languages other than Pascal, because FreePascal's progenitor Turbo Pascal has dominated the Pascal world for so long that most of the earlier dialects have simply disappeared. If you ever attempt to program in different versions of BASIC, on the other hand, the dialects problem will appear in spades, and you will have a great deal of work to do making programs written for one BASIC compile correctly using another BASIC.

In general, this book will be speaking of the FreePascal dialect of Pascal. (FreePascal speaks several, as I'll describe later on.) Here and there, I'll be pointing out differences between FreePascal and other Pascals, but as I've said before, the differences are becoming less and less important as time goes on.

2.2. Operating Systems and User Interfaces

When I wrote the first version of this book, way back toward the end of 1983, computing was a much, *much* simpler field. Most people who had personal computers had IBM PCs. Fewer people (but still a lot of them) had Apple IIs. Some further number had an odd assortment of machines from many manufacturers that had little in common but an operating system called CP/M. The Macintosh did not yet exist, nor did Microsoft Windows.

Things are different now, let's say.

The operating system called DOS was not an "operating system" as the industry defined it in the 1980s. At best it was a sophisticated file manager, and most of what it did was write files to disk drives (generally floppies until the midlate 1980s) and read files from disk drives. "Running a program" was just another file operation: DOS read a program file from a disk drive and stored it in RAM, and then stepped back and gave control of the CPU to the program that it had recently placed in RAM. When the user exited the program, DOS would regain control of the machine— with some luck. My 1979 CP/M machine would literally reboot itself each time a program exited, so that CP/M could again regain control. ("Rebooting" was a far simpler process with those ancient machines, and took only a second or two.)

What was true of DOS was also true of CP/M and the Apple II operating system. Only one program could run at a time. There was no networking without a great deal of added complication. The user interface was a "command line" on a text-only screen that could at best display 80 characters wide by 24 characters high. To run a program you had to type a command at the command prompt and then press Enter. "Graphics mode," where it existed at all, was a slow and limited special case. There were no windows, no task bar, no mouse, nor any drop-down menus except as provided by the occasional forward-looking program.

The text-mode command-line user interface was the default until the late 1980s, when personal computers finally became fast enough to manage images on a graphical screen. The Macintosh was the first successful graphical user interface (GUI) machine, but I remember sitting in front of Bruce Schneier's brand-new Macintosh in late 1984 and feeling like the poor little thing was struggling terribly. Microsoft Windows was virtually unusable—and generally ignored—until 1990.

Although I adopted Windows NT when it was released in 1993, Windows itself did not become generally accepted until Windows 95 appeared in 1995. The DOS command-line interface was widely used until Windows 98. It wasn't until 2000 or so that most people had come to depend on graphical user interfaces, and about that time the whole idea of a command line and text screens was mostly forgotten, except by mainframe programmers and Unix users.

A fifth grader once wrote on an exam: "Now that dinosaurs are safely dead, we can call them clumsy and stupid." It's easy to dismiss the DOS-style command-line interface as clumsy, awkward, and time consuming, but it has the virtue of simplicity. (We also forget that dinosaurs ruled the Earth for 100 million years. They must have been doing *something* right.) It's also far from extinct.

Text mode and console windows

Command-line interfaces are still used, as are text-mode displays. Although non-technical people rarely use it, Windows contains something called the "Command Prompt," (look in the Accessories group) which is basically a DOS prompt in a black window that displays only text characters. Much ancient DOS software will still run if executed from the Windows command prompt. More to the point, some crucial Windows utilities *only* run from the command prompt, and only accept textual commands and output only text lines.

Figure 2.1. A console window under Windows 2000

The window in which the command prompt appears is generally called a *console window*. Figure 2.1 is a typical console window as displayed by Windows 2000 and after. The desktop versions of Linux and Unix also have console windows. Mac OS/X has a console window called Terminal. All console windows on whatever operating systems work and look pretty much the same. (The group of commands that each understands is different, of course.)

The existence of console windows make certain things easier for people creating computer language compilers like FreePascal. If a compiler only communicates with its users through simple text screens, it can work pretty much the same way

no matter what operating system is in control of the computer. So it was done with FreePascal: By default, it works in a console window. This means that you can open a console window and type textual commands to FreePascal, which responds by displaying textual information in the same window on the next line or lines. Figure 2.2 is a console window "conversation" with FreePascal, as it would occur during the compilation of a very simple pascal program.

Figure 2.2. A FreePascal compilation operation in a console window

All the major FreePascal implementations look pretty much the same when run in console windows, whether you're using Windows, Linux, OS/X, or anything else. Of course, once you begin writing programs that work specifically with a graphical user interface belonging to a particular operating system, the explanations begin to diverge. I won't, however, be going that far in this introductory volume.

Text-mode user interfaces

Output in a console window is generally line-by-line: A program displays a line of text and then moves the cursor down to the next line below it. However, that's not all that can be done. With a little bit of additional software behind it, a console window can be treated as a grid of character locations, and addressed by X,Y coordinates. Text can be displayed anywhere on a console window, at random, under program control, by specifying an X,Y location at which to begin text display. Text can be displayed in different colors. Boxes and frames may be created using special characters. (IBM pioneered text-mode "graphics" this way with their original 1981 IBM PC.) Furthermore, the console window text cursor may be directed around the window by the computer's mouse.

Abilities like these allow us to create simple editing and compiling environments entirely in text mode, to run in a console window. Many of the concepts we're used to in graphical operating environments like Gnome, KDE, and Windows can be emulated, albeit on a character basis and with far less resolution. Such an environment (whether implemented in text-mode or graphics mode) is called an *Integrated Development Environment* (IDE), and one is included with and installed with FreePascal on most operating systems for which the compiler is available. The FreePascal console window IDE is shown in Figure 2.3, with a simple Pascal program loaded and displayed, ready to be compiled and run.

```
 File   Edit   Search   Run   Compile   Debug   Tools   Options   Window   Help
[ ]══════════════════ D:\FreePasQ1\EatAtJoes.PAS ═══════════════════1=[ ]
PROGRAM EatAtJoes;

USES Crt;

BEGIN
  ClrScr;
  GotoXY(15,11);
  Writeln('Eat at Joe''s!');
  GotoXY(15,12);
  Writeln('Ten Million Flies Can''t ALL Be Wrong!');
  Readln;
END.



      1:1
 F1 Help   F2 Save   F3 Open   Alt+F9 Compile   F9 Make   Alt+F10 Local menu
```

Figure 2.3. The FreePascal Text-Mode IDE in a Console Window

Graphical IDEs for FreePascal programming

FreePascal's text-mode IDE is simple and can be very handy, but the software is a little fluky and doesn't always work correctly on all operating systems. Fortunately, you aren't limited to the text-mode IDE that comes with the compiler. Graphical IDEs exist for FreePascal, and there's no reason not to use one if one is available for your OS .

The best of these is Lazarus, which at this writing (2021) is available for Windows, Linux (including the Raspberry Pi OS), FreeBSD, Mac OS/X, and others. Lazarus is an ambitious Rapid Application Development (RAD) system very much in the mold of Borland's (now Embarcadero's) Delphi. Figure 2.4 shows the Lazarus RAD environment in its full glory, ready to create windowed GUI applications.

Lazarus is a complicated system and was created specifically to develop fully event-driven windowed GUI applications like the one shown above, but it's perfectly at home managing very simple Pascal programs that communicate through console windows. It may seem like overkill for the sort of teaching
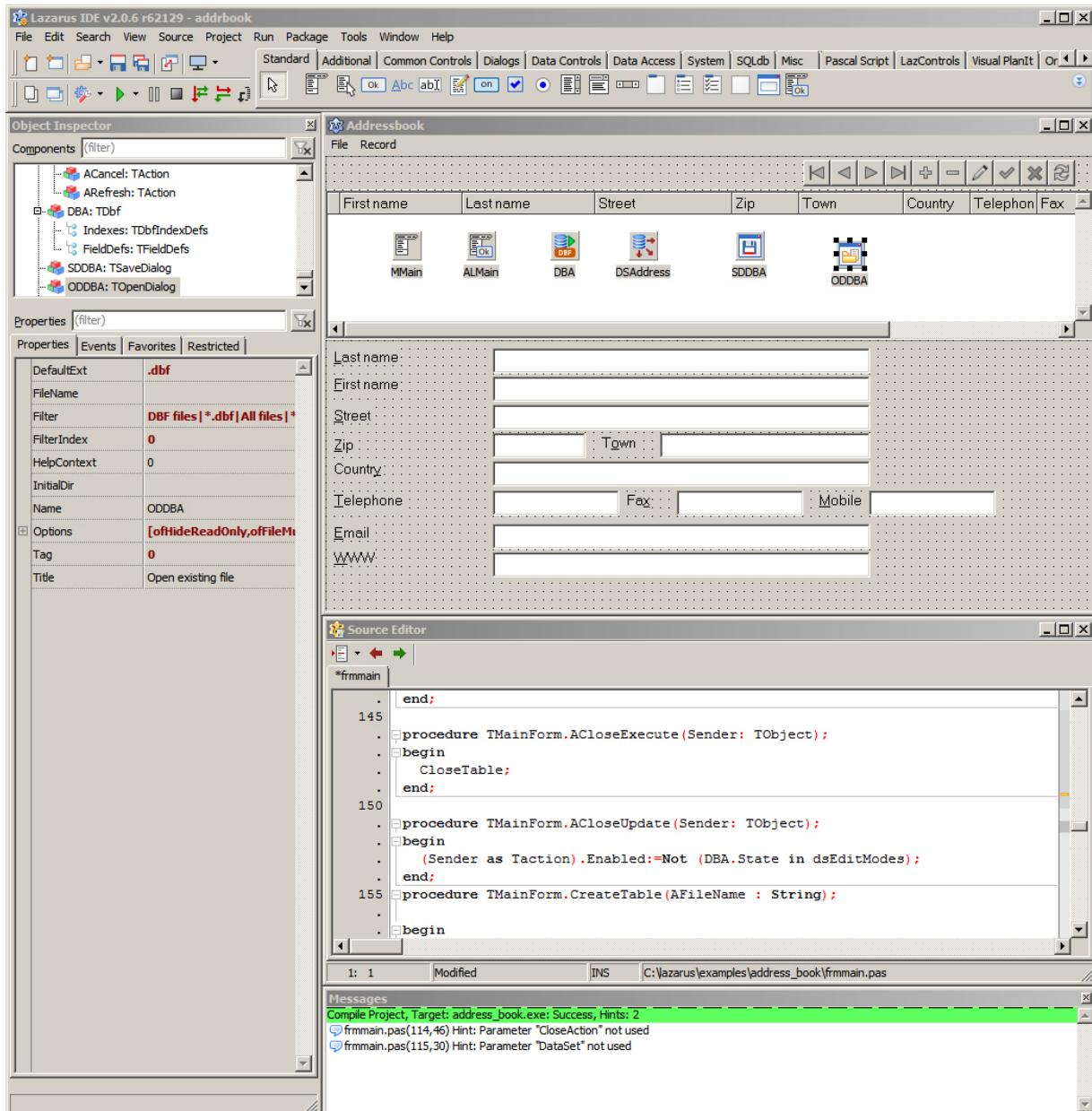
Figure 2.4. The Full Lazarus RAD Environment in Action

programs I'll be presenting in this book, but when you've learned enough Pascal to move up to object-oriented programming and the creation of windowed graphical applications, you'll find that Lazarus makes the process *hugely* easier. That's why I want to use Lazarus as the teaching IDE from the very beginning. The skills you develop now by using Lazarus to write simple programs will make learning the rest of it a snap later on when you need it.

Lazarus is itself a multi-window application, and the good news is that we can make Lazarus a lot simpler to understand by closing the windows that you don't need for the time being. Figure 2.5 shows Lazarus as it appears while you're writing a simple Pascal program. The program code is edited from a GUI window (here, under Windows XP) but the program's output is displayed in a console window.
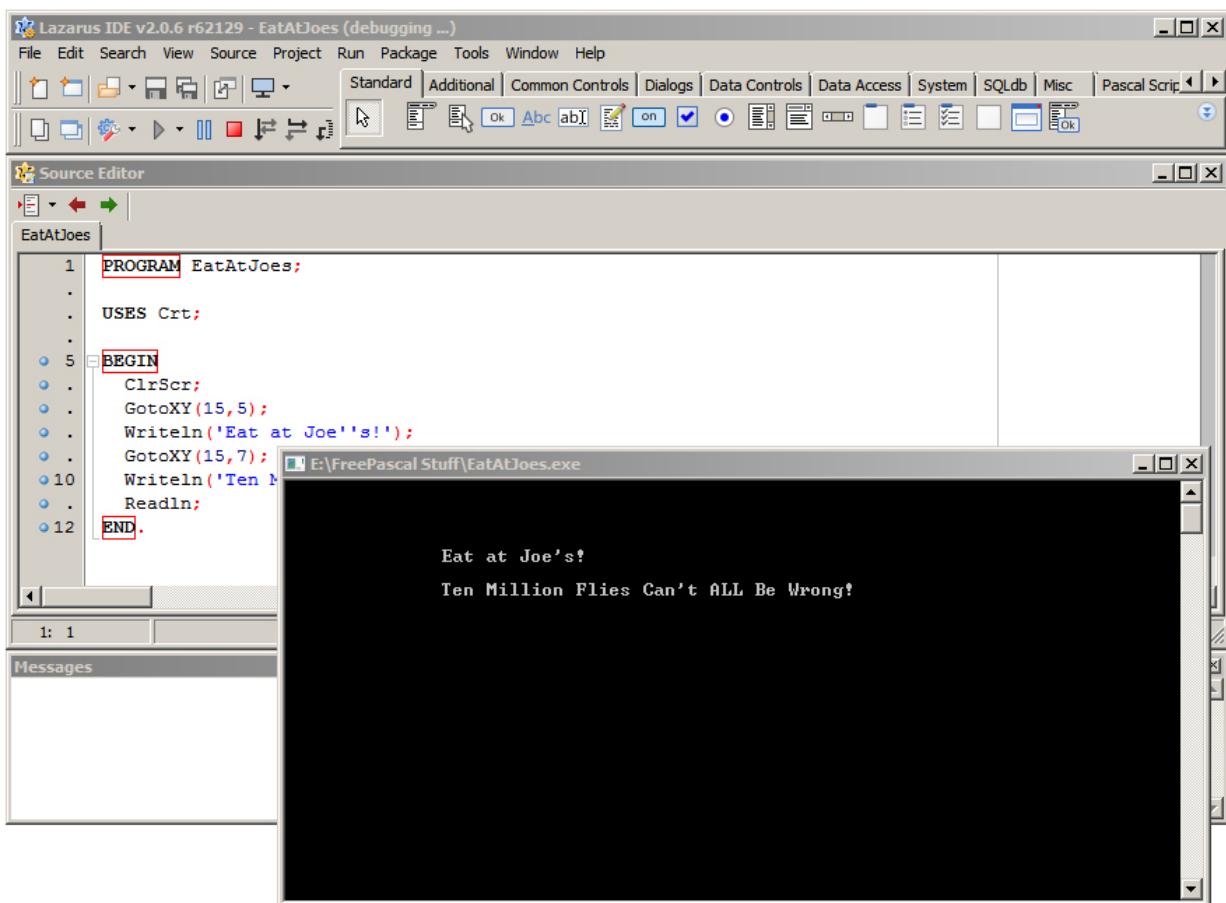
Figure 2.5. Lazarus Used to Edit and Run a Simple Pascal Program

There are four Lazarus windows shown: The control window is the narrow one at the top. It's the "boss" window that controls all the rest of the Lazarus system. The larger window in the center is the editor window, where you write your Pascal code and fix your inevitable errors. The slim window at the bottom is the messages window, which tells you what the system is doing and points out your errors. The black window is a console window, and is not itself a part of Lazarus. Lazarus launches a console window when your program runs, to provide a separate "blackboard" (literally!) on which your program can write its output, and display the input that you enter through the keyboard.

2.3. The FreePascal Process

FreePascal is itself a program, and needs to be installed on your hard disk before you can begin to learn how to use it. Later on, I've devoted an entire chapter to the installation of FreePascal and Lazarus so as not to get bogged down in installation details right here. Later in this chapter I'm going to run through the compilation of a short, simple program. You don't necessarily have to run FreePascal to benefit from that run-through. Reading it carefully will be sufficient. I'm simply trying to get you familiar with the shape of the process of creating programs with FreePascal. However, if you're a practical, hands-on kind of person, you might want to stop here, jump to Chapter 4, and install FreePascal and Lazarus on your machine if you haven't already. That way, you can come back to this section and follow along with a real program while I'm explaining what's going on. You don't need to go through the rest of this chapter to install the software. Now or later: It's your choice.

Note well that I'm going to go into the mechanics of using the FreePascal compiler and the Lazarus Environment in much more detail later on in this book. The overview in this section will be very brief.

Source files, object files, resource files, and project files

Two words you may have heard before and will certainly hear in the future are *source code* and *object code*. Source code files are the "human readable" form of a program. A source code file is an ordinary text file that you create by typing into a text editor like the one built into Lazarus. (You can edit source code files in any text editor; people have used Windows Notepad, Wordpad, or even Microsoft Word; and there are a multitude of text editors available for Linux.) Object code files, on the other hand, contain the individual machine instructions that the CPU knows how to execute. These are those endless runs of 0's and 1's that I described in the previous section. The whole purpose of the FreePascal compiler is to take source code files that you write, and use them to generate object code files that may be executed on a computer.

Source code files for FreePascal usually have a file extension of .PAS. Object code files come in several types. *Unit* files contain bits and pieces of programs that can be used again and again, but are not in themselves complete programs. They're basically libraries of precompiled code. *Executable* files are real programs that you can run from your operating system.

For decades, programming was simple enough so that source code files and object code files were all there were. Today, the demands of GUI programming have given birth to yet another class of files: *Resource* files contain things like icons and

images that must be "baked into" an executable program file. Finally, there are files associated with a programming project that summarize the details of the project for the benefit of the programming environment itself. In our case, that would be Lazarus, but all of the ambitious RAD environments used today generate *project* files to manage the mountain of details inherent in advanced programming projects. You may not have to pay details attention to resource and project files while writing simple programs in FreePascal, but at least understand that they're necessary. For the most part they are generated and maintained by the RAD environment, and while they must be understood, you don't often have to explicitly edit them.

In your early explorations of FreePascal, you'll be creating simple, standalone .PAS files and compiling them to executable files. (I'll explain about creating unit files for maintaining reusable code libraries later on.) You'll then run the executable files in a console window to see how well they work. The executable files may be run from a console window prompt without any help from the FreePascal compiler or the Lazarus IDE. However, for the example programs in this book you'll generally be able to test your executable files from inside console windows created by the Lazarus IDE itself.

Cross-platform programming

This is a slightly advanced topic, but it's worth noting as part of the big picture: FreePascal can handle *cross-platform* programming. What this means is that you create a program on one operating system running on one type of computer hardware, but generate programs that can run on other, different operating systems or hardware. For example, if your main machine is Microsoft Windows running on Intel x86 hardware (a very common combo that we often call the "Wintel" platform) you can still write programs that will run under Linux, Unix, Mac OS/X, and a long list of other operating systems that you may not have even heard of.

There are often limitations on what such programs can do. For example, there are whole classes of operating systems that don't have a graphical user interface. Everything that they do is done in a purely textual environment much like a console window. If you load a graphical windowed program intended for the Linux GNOME windowing environment on such a system, the machine will throw up its hands in despair because it won't know how to deal with requests to create windows or accept mouse clicks to buttons.

The issues that come into play during cross-platform programming are many and they are subtle. You need to become fluent in FreePascal long before you should attempt to create code on one platform to run on another.

The program development cycle

Creating programs with FreePascal works like this: You begin by conceptualizing a design of some sort for your program. This might involve some study, some notes, and some drawn diagrams indicating how the program is to work. With your design (that is, your notes and diagrams) close at hand, you bring up a text editor window and begin writing program code based on your design into the code editor window. For the tutorials in this book, that editor window will be part of the Lazarus IDE. (You can use other text editors if you want to, and many people do.) Every so often during this process, you must save your program source code file or files to disk. For the Lazarus IDE that's as easy as pressing the Ctrl-S key combination.

Once you've completely typed your program's source code file into the text editor window, you invoke the FreePascal compiler. This, like most of the commonly used commands, can be done by pulling down a menu option or by pressing a key combination. (In this case, it's just a single key, F9.) The compiler reads the source code that you've typed into the text editor window and generates the appropriate object code files, plus a single executable file.

Once the compiler begins running, one of two things happens: The compiler either finds an error, or it doesn't. If the compiler does *not* find an error, you have what is called a *correct compilation*. This doesn't mean you have a bug-free program yet by any means—but we'll get to *that* little matter in a moment.

Most of the time (especially while you're still a Pascal newcomer) the compiler will "complain" about something in your source code. You may have typed something incorrectly, or else misunderstood some element of Pascal and written something incomplete or nonsensical. No object code files will be created in this case. You'll have to stare at your code a little more, read the online help (or this book!), correct what's wrong, and try the compilation again.

Sooner or later, your program will compile correctly. At that point, you'll have object code, including an executable file, and you can try running it from within the Lazarus IDE.

As with compilation, when you try running your program, one of two things happens: Either it works perfectly, or else it doesn't. And when it doesn't work perfectly, that's when we say that your program has those legendary *bugs*.

Getting rid of bugs is a lot tougher than just getting rid of compiler errors. The compiler will usually give you strong hints about where a compiler error lies and what's causing it. Bugs, by contrast, range from some simple, innocuous action that you didn't ask for (or one you asked for that didn't occur) all the way to provoking the operating system to abort execution of your program and throw it out of memory. The compiler doesn't spot bugs. *You* do.

FreePascal and Lazarus contain a number of built-in tools to help you flush out the inevitable bugs you'll find in your programs. Using these tools and your own good sense, you gradually find and fix the causes of whatever bugs come to light. This process can take awhile. Getting rid of disastrous and obvious bugs happens early in the cycle, because you're pretty motivated to find them and fix them. Getting rid of minor or subtler bugs that don't necessarily make your program worthless could be a long process—and for programs of a useful size may be an endless one. People say that there's always one more bug, and you can devote as much time and energy as you care to rooting that last bug out.

But even when you root the "last" bug out, *there's always one more bug.* Trust me.

The process, summarized

At this point, let's run down a list of the major steps in the FreePascal program development process:

1. You design the program on paper. This does not mean simply writing out program statements with a pencil. It usually means charts, diagrams, and high-level notes we call *specifications.*

2. Working from your design, you type program source code into a text editor window. (This will generally be within the Lazarus IDE.) Save your code early and often!

3. Once you consider the source code complete, try to compile it.

4. Fix any compiler errors that come up in the text editor window, save the edited file, and then recompile to see if you've fixed any errors. Repeat the process until there are no more compiler errors.

5. Once the program compiles correctly, try running it. Take note of any bugs that appear, where a bug is anything a program does that it isn't supposed to, or something it doesn't do that it should.

6. Fix all identified bugs, and run the program some more, to see if you can identify any further bugs. This stage is called *testing.* It takes a long time.

7. When you can't find any more bugs, the program can be considered finished. This doesn't mean that the program doesn't contain any more bugs. It generally means that you've simply run out of patience with bug-chasing, and will be content with what you have. Days, weeks, or months later, you may become sufficiently irritated by one bug or another to begin the debugging process all over again, starting with Step 5.

I've summarized this process in Figure 2.6.

Compile

Edit   Compiler Errors   Test

Identify Bugs

Program Design

No Bugs

Start Here   You're Done!
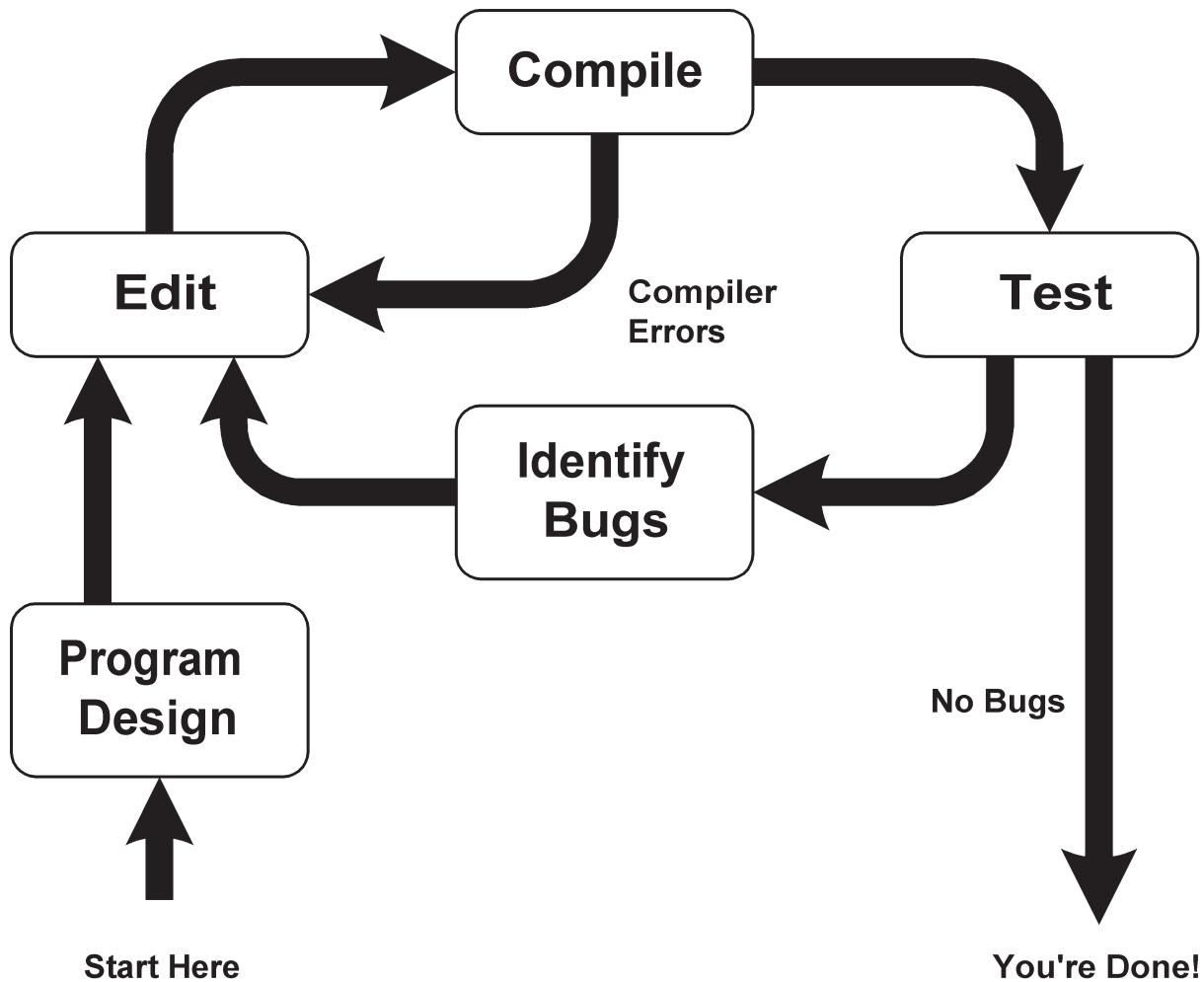
Figure 2.6. The FreePascal Programming Process

2.4. LET'S TRY IT!

With all that under your belt, it's time to see the actual compilation process in action. I'd like you to type in a simple program, save it, compile it, debug it, and run it. I'm skipping the design phase for the time being, since I've found it's difficult to explain the design process to a person who isn't yet comfortable with the programming language itself. This doesn't mean that *you* should skip the design phase later on, or dive into writing code before giving your design the attention it deserves. It's just that for the purpose of learning the software development process, you need to know a little about coding first.

To actually follow along with me, you'll obviously have to install FreePascal and the Lazarus RAD environment first. I describe the installation process in detail in Chapter 4. I encourage you to turn to Chapter 4, find the installer files, download them, and install them on your system.

Before you launch Lazarus, create a working directory somewhere on your machine. How this is done varies by operating system, but it doesn't matter where it is as long as your filesystem allows you to create and change files there. The directory you create here will be used to store the programs that you enter and compile.

Now run Lazarus itself, however that's done on your particular machine. For Windows this means you double-click on the Lazarus icon, assuming that the installer created a desktop icon. You can also run Lazarus by navigating to it in the Start menu. How software is launched under the Linux shells depends on the shell and how you've set up launchers and panels and so on.

Once Lazarus is running, create a new project by selecting **Project | New Project** from the Lazarus main menu, and selecting **Simple Program** from the **Create a New Project** dialog that appears. (The dialog lists all the various sorts of projects you can create with Lazarus.) Assuming that you've configured it as I describe in Chapter 4, what you'll see will look a great deal like the screen shown in Figure 2.7. (Much of that configuration involves hiding the parts of Lazarus that you don't yet need.) The label "project1.lpr" is the default name for a new project that you have not yet given a name to. We'll give it a proper name shortly.
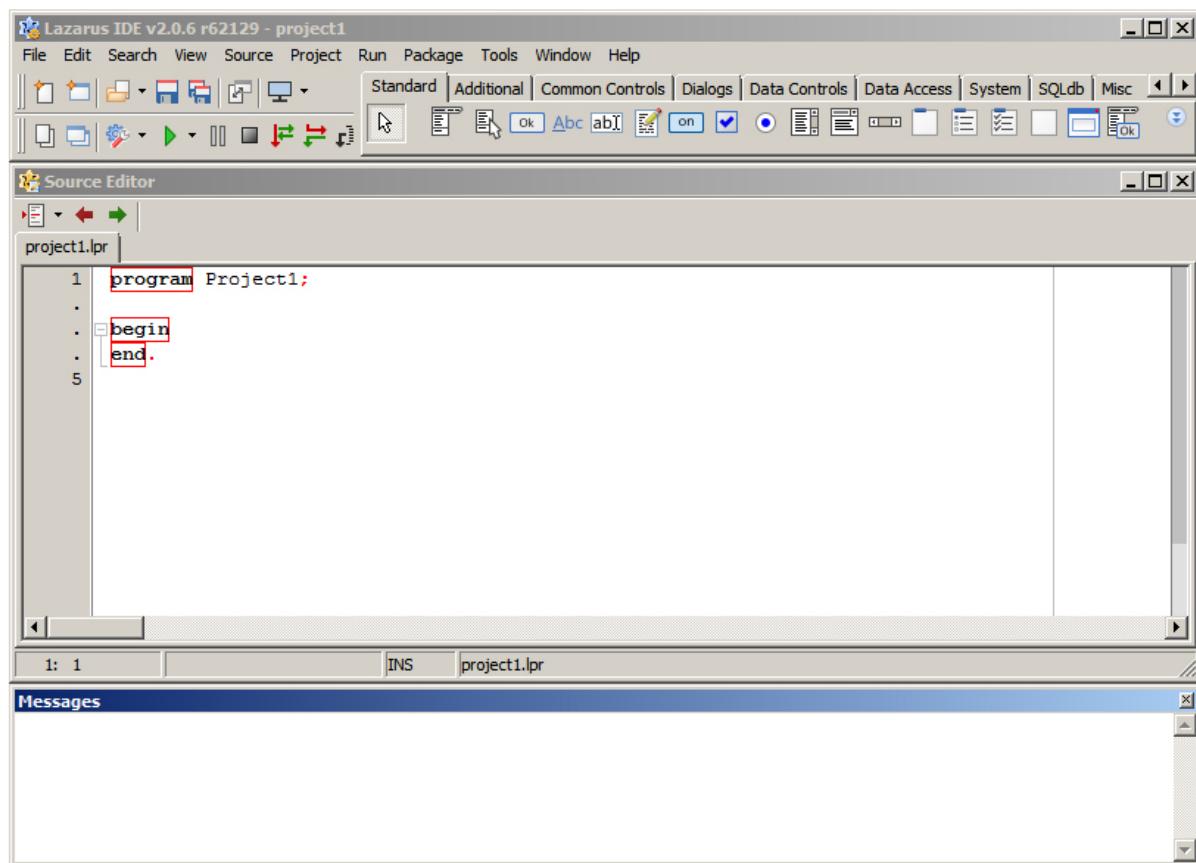
Figure 2.7. The initial Lazarus IDE screen

Entering a program to an edit window

An edit window is already open and ready to go—it's the mostly-white window labeled "Source Editor." However, something's already there. In creating a brand-new project, Lazarus has created a sort of skeleton program for you. The program is technically "empty" in that it contains no Pascal statements that do anything useful or even visible. It is, however, a complete framework for a Pascal program. All you need to do is add the Pascal text that define the program's action.

The test program is shown below in its entirety. Some of it was already "typed in" by Lazarus when it created the new skeleton program. The words **PROGRAM**, **BEGIN**, and **END** are all there in black, albeit in lower case. (I'll talk about the character case issue shortly.) Your job right now is to enter all of the text in the test program that is not already present in the editor window.

```
PROGRAM EatAtJoes;

USES Crt;

BEGIN
  ClrScr
  GotoXY(10,5);
  Writeln('Eat at Joe''s!');
  GotoXY(10,7);
  Writeln('Ten Million Flies Can''t ALL Be Wrong!');
  Readln;
END.
```

Here's how to proceed:

1. Highlight the default project name after "program" and replace it with "EatAtJoes".

2. Add the line **USES Crt;** including the semicolon. The **USES** clause contains a list of code libraries, and **Crt** is the only one this particular program requires. More complex programs will require more.

3. Enter the text I show between the words **BEGIN** and **END** into the editor window, between **begin** and **end**. precisely as shown.

The character case issue

Before we go any further, I must emphasize that, in Pascal, *character case does not matter to the compiler*. In other words, FreePascal sees **BEGIN**, **Begin**, and **begin** (or, for that matter, **bEGin)** as *precisely* the same thing. Unfortunately, Lazarus's defaults and I differ on a key formatting issue: I feel very strongly that Pascal's reserved words

like **BEGIN** and **END** should be in uppercase only. The default in Lazarus is all lowercase. So in any code that Lazarus generates automatically (like the skeleton program here, and much else in more complex programs) reserved words will be in lower case. This is a bad idea. Although you can do it whichever way you like, keep this in mind: *Reserved words are different.* They have "super powers" and must be treated specially. It pays to set them off somehow from ordinary program identifiers like constants and variables, to make your program code more readable and less vulnerable to certain types of completely avoidable errors.

Some people will try to tell you in a weirdly excitable tone: "But...but...uppercase letters mean that you're...*shouting!*"

Maybe somewhere in the world they do, somewhere ancient and obsolete but alas, not yet dead. In Pascal, reserved words are the framing members of your programs. *They must stand out.* If you put them in lowercase, you will miss them now and then and make stupid mistakes, and spend more time and energy fixing things than you otherwise would.

Fortunately, there's a way to configure Lazarus to keep reserved words in uppercase. Go to Tools|Options|Codetools|Words. Under Keyword Policies you can select one from several radio buttons. Click UPPERCASE and then OK. Now, when Lazarus uses a reserved word it will be in uppercase.

I'll speak in more detail of reserved words in a later chapter.

Giving a name to your new project

At this point, you've entered the text for the test program, but you're still using the default name that Lazarus gave to the project on its creation. Giving a descriptive name to the project is our next step.

To name the new project, select **Project | Save Project As**. (There's no hotkey.) A dialog will appear allowing you to enter a new name for the project. Leave the default file extension as it is. In this case, call the project "EatAtJoes" and click **Save**. Correctly entered and properly renamed, your project will look like what's shown in Figure 2.8.

Saving your work to disk

It's hard to overemphasize the importance of saving your work to disk frequently. For a certain period of time, what you've just typed into the open edit window exists only in RAM memory *and nowhere else.* If your machine failed for some reason, or if lightning hit a power pig and knocked out the electricity to your neighborhood, what you just typed would be gone for good. Such irritations are never necessary.
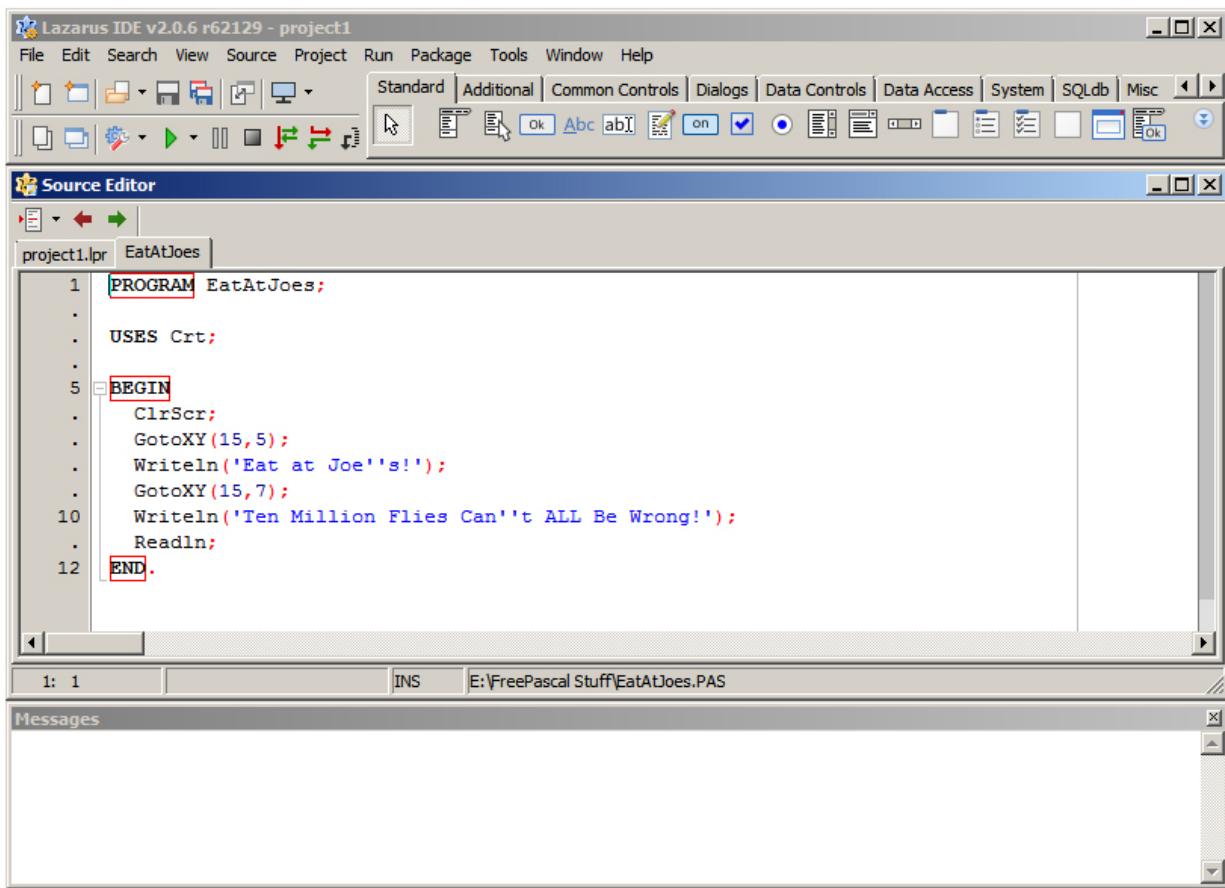
Figure 2.8. The project entered and renamed.

Get it down on disk regularly by saving the project.

In Lazarus, this is almost trivial. Just click the Save icon (which looks like a diskette) in the main menu bar (fourth icon from the left) or press Ctrl-S. Lazarus will save your project into your working directory. If you haven't already given your project a name, Lazarus will pop up the **Save Project As...** dialog, and allow you to specify a directory and a name for the project. Thereafter, saving the project will be done without a dialog, and it will be saved to the directory and name you specified the first time you saved the project.

There's more to a project than just one file. If you look at your working directory after naming and saving your project, you'll see several files with your project name and different extensions; for example, `EatAtJoes.lpr`, `EatAtJoes.lpi`, `EatAtJoes.lps`, and so on. The file `EatAtJoes.lpr` contains the Pascal source code you just typed in. Once you build the project there will be even more. Don't worry about what all the files are nor what they do right now. I'll have more to say about the various files that make up a project in Chapter 5.

Compiling and building your program

You've now got a new program typed in and saved on disk as a Lazarus project. It's time to turn the FreePascal compiler loose on it. This step will discover any errors you might have made while typing, and if the program is correct, it will create (among others) an executable code file that can be run and observed.

At this point I have to refine the jargon a little: Compiling a source code file, as I explained earlier, generates an object code file from the source code file. However, creating an executable file that you can actually run isn't always done as part of the compile step. In years past, when computing was simpler, Pascal compilers did generate an executable file when compiling a program. Today, however, things have gotten complicated enough that executable files are not generated in one pass. Even fairly simple programs may consist of more than one Pascal source file. Each of these files must be compiled, and then the several object code files then linked into a single executable file. The whole process taken together—compiling all the various Pascal source code files that are part of the project and then linking them into a single executable file—is called *building*. We use the term even for the simplest projects that have only one single Pascal source code file. We can *compile* a single Pascal file, but we *build* a Lazarus project.

Building your project is done from the **Run** menu. Pull down the **Run** menu and select the **Build** option. Lazarus will launch the FreePascal compiler behind the scenes, and do whatever else needs to be done (which varies among different computing platforms) to generate an executable file. On such a small program file as this first example, FreePascal does its work very quickly.

And you'll have to forgive me, but I slipped a minor error into the Pascal file as I printed it on page 58. If you typed the code into the editor window exactly as shown, FreePascal will post an error message in the Lazarus Messages window. Of course, if you know a little about Pascal, you might have assumed it was a typo and fixed it yourself. If so, "unfix" it and select **Build** again. Your screen will then look like Figure 2.9.

Spotting and fixing compile-time errors

This is what we call a *compile-time error*, because it turns up during the compilation process that occurs doing a project build. It's a very common one, not only for newcomers to Pascal but for us old-timers as well. Missing semicolons will just happen now and then, no matter how much of an expert you are. There's a science to semicolon placement, which I'll take up in detail later on in Section 9.8. Trust me this time: I should have placed a semicolon at the end of the line **ClrScr**.

Notice from the error message that the compiler is giving you a hint, but also that it's not telling you exactly where the semicolon is supposed to go. The compiler is
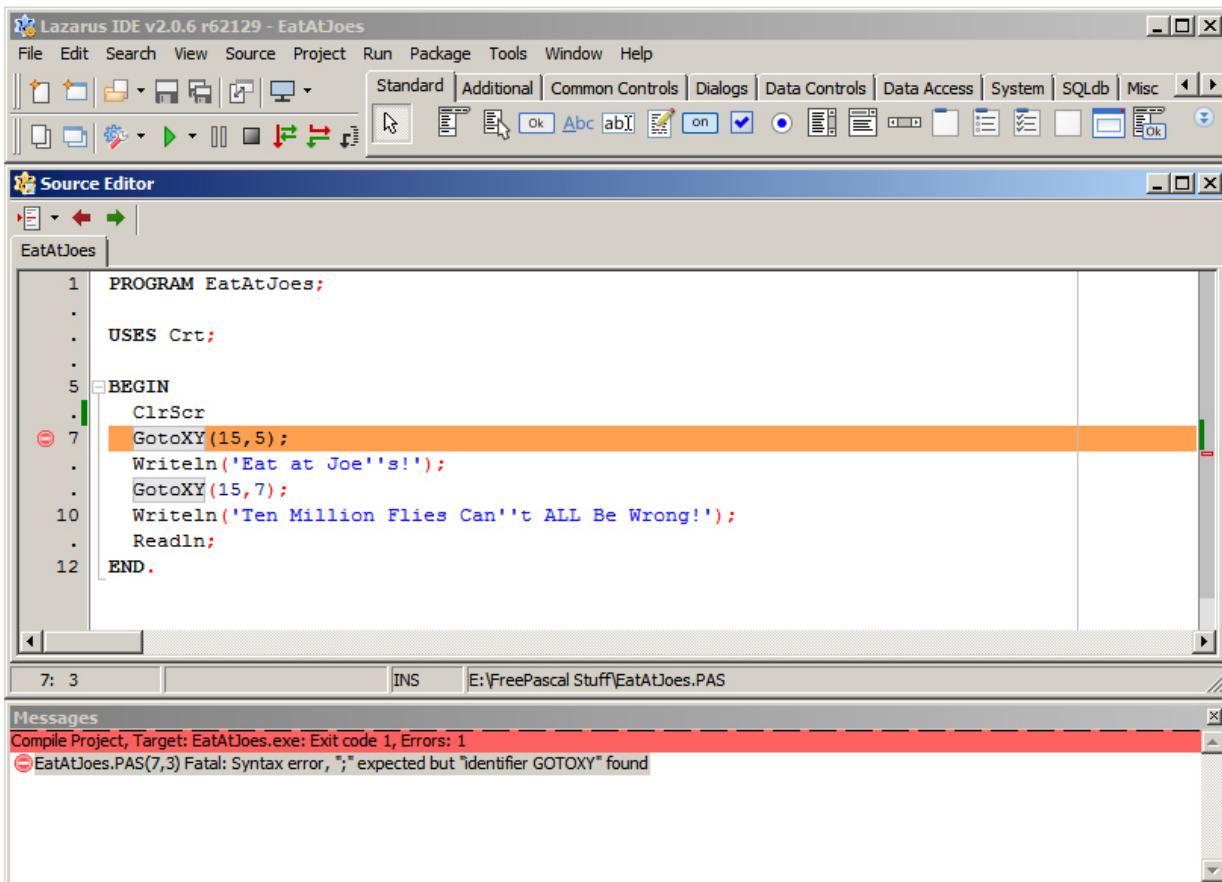
Figure 2.9. A FreePascal compiler error message.

smart, but it has limitations. One of these is that it will point out an error not where the error actually is, but where the compiler first noticed that there was an error.

It's a little like the situation you find yourself in when you go out after supper to fill your gas tank and forget your wallet. You made a mistake when you failed to put your wallet back in your pocket when you changed pants after work. But you only notice the mistake when you've pumped a tank full of gas and reach for your credit cards. Whoops...

I make the point here because a lot of people assume that FreePascal not only discovers errors but points out where those errors are. Not true—it can only tell you where it first noticed that something was wrong. Keep that in mind as you struggle through your first few error-rich sessions with FreePascal!

Repairing this particular error is easy. Place the text cursor at the end of the **ClrScr** line, and type a semicolon there. Click the Save icon (the diskette) to save the change. Then select **Run | Build** again.

Or, if you think there may be still more compile-time errors lurking in your Pascal code, select **Run | Quick Compile**. This option is different from Build in that it only compiles the source code file currently loaded in the Source Editor window. When you're working on a large, multi-file project, Build will take more time. If you're simply checking the current source code file for errors, **Run |Quick Compile** will do the job faster. (For small files like this you won't see much difference.)

Lazarus doesn't blow a trumpet or pop up a special window when a build completes successfully. All you'll see is the following message in the Messages window, in very small type:

```
Project "EatAtJoes" successfully built
```

There will be one more line displayed in the Messages window:

```
Options changed, recompiling clean with -B
```

All this means is that Lazarus used with "-B" command-line option when it invoked FreePascal to build your project. The -B (Build) option creates a "clean" compile in which all Pascal files in the project are compiled fresh, and no object-code files generated earlier are used. There's only one Pascal file in this very simple project, but in more complex projects you have the option of compiling only those Pascal files that have changed since the last time you compiled them. This can save you a considerable amount of time on very large projects with many source code files.

Running your program

When you see the message above, the compile pass was correct, the build was accomplished, and you've now got an executable code file out there somewhere, waiting to be run. You don't technically have to know where it is, or even what its name is. The name is easy, though: `EatAtJoes.exe`. The executable version of a Pascal program created with Lazarus and FreePascal is the same as the project name, but with .exe as the file extension rather than .lpr.

Executing your programs is also simple: Click the right-pointing green arrowhead in the second row of icons in the main Lazarus menu bar. Your program will be executed, and our backhanded advertising slogan will be displayed in a console window. The slogan will remain on the screen as long as you choose to leave it there. To end program execution and return to Lazarus, press the Enter key.

The arrowhead icon in the main menu bar isn't the only way to execute your program. You can also select **Run | Run**, or simply press the F9 shortcut key. Note also that to run, your program does not require Lazarus or FreePascal at all. In your

working directory (or wherever you saved the EatAtJoes project) you'll find a file called `EatAtJoes.exe`. This is a complete and independent executable file, and you can run it by naming it on a console window command line, or by double-clicking on it in Windows Explorer. (Other graphical environments may have their own ways of executing standalone executable files.)

An interesting point to be made here is that FreePascal, Lazarus, and the EatAtJoes program you've just successfully run are all native-code programs, conceptually identical and pretty much equal in the eyes of the computer itself. The programs you're writing are not "toy" programs by nature nor are they crippled in any way. (They're just small—for now.) You could theoretically write something as complex as—or even more complex than—FreePascal itself. This is definitely big-time programming. Don't let anyone ever tell you otherwise.

When the program that you ran finishes its execution, it immediately hands the baton back to the Lazarus environment. You can then continue the development cycle of write code, save, build, and run.

2.5. RECAPPING DEVELOPMENT BASICS

As you might imagine, what you've just seen barely scratches the surface of what FreePascal and Lazarus can do. But for the first few small programs you'll write with FreePascal, that's just about all you really need to know about it. The mechanics of elementary FreePascal development come down to this:

1. Create a new project. Enter your source code into the Source Editor window, expanding the skeleton file that Lazarus creates when it creates a new project.

2. Save your project (which saves your source code) under a new, meaningful name by selecting the **Project | Save Project As...** menu option. It's a good idea to save a new project before you've typed more than a screen full of source code—and save it often (by using the Ctrl-S shortcut key or clicking the Save icon) after you've saved it that first time.

3. Build the project using the **Run | Build** menu option. The build process identifies any compile-time errors in your source code files. If no errors appear during a compile, your program is compilable and correct. (This doesn't mean it has no bugs!) You can also build the project using the Ctrl-F9 shortcut key.

4. Test your new program by clicking on the green arrowhead Run icon, or by selecting the **Run | Run** menu option. You can also run your program using shortcut key F9.

Here are a few other points worth remembering:

- As much as you might be tempted to do otherwise, spend some time designing your program before you begin to write actual program code in the FreePascal IDE editor. We'll talk more about program design later on.

- Character case does not matter. **BEGIN** and **begin** are exactly the same!

- Text shown in bright red in the Source Editor window are instructions to the compiler, not Pascal code! Lazarus generates some of these instructions automatically, and as you become more of an expert and write more complicated programs, you may find it necessary to add some to the project yourself.

- When a problem comes up during compilation, it's called an error. When a problem comes up during your actual testing of a program that has compiled correctly, it's called a bug. The compiler will give you some hints when it discovers an error. But you're pretty much on your own to identify and correct bugs.

Finally, this would be a good time to go back and take one more look at `EatAtJoes.lpr`. It's a series of steps bracketed between the two words **BEGIN** and **END**. In this simple program, these steps should be close to self-explanatory, even thought you may never have looked at a single line of Pascal code before.

The line **USES Crt** simply tells FreePascal to use a code library named **Crt**. We'll talk about using code libraries later on in this book. The only other line that might have you (as a beginner) squinting hard is the last program step: **Readln**. That's the step that waits for you to press the Enter key so that the program can terminate its execution and give control of the console window back to FreePascal. As with all of the fundamental elements of FreePascal code, we'll return to **Readln** in more detail later in this book.

CHAPTER 3.
THE SECRET WORD
IS "STRUCTURE"

OK. Do you want to build a shed? Or do you want to build a cathedral? With FreePascal (or with most any programming language, actually), *the choice is yours.* And the difference (far more than simply scale, since there can be mighty big shacks and mighty small cathedrals) is solely a matter of *structure.*

You may have heard this before. It's been said many times, and (far too often) said badly. People often lose track of the difference between small-scale knowledge and big-picture knowledge, and the corollary difference between small-scale quality and big-time mess. You can perfectly memorize the usage of every single reserved word, operator, and predefined identifier in FreePascal, and still not have the least clue as to how to write a program with any hint of quality. Smearing individual statements around on the screen can be fun, and such smearings can actually compile and (sometimes) run, but hey, how much fingerpainting hangs in the Louvre?

In this chapter, I'm going to provide a sort of "view from a height" of FreePascal program structure. Along the way, you'll pick up some of the fundamentals of defining and using variables, as well as some of the simpler operators and standard functions. It's tough to explain structure when you haven't yet explained the boards, bolts and girders from which the structure is made!

Note also that almost everything I explain in this chapter is true of almost any dialect of Pascal you'll ever see. What we're exploring is the fundamental nature of the language itself, quite apart from any individual implementation. *Pascal is about organizing complexity into comprehensible form.* Keep that in mind. If you don't, you might as well work in mud-pie languages like C and C++.

3.1. TAKING IT SUBSYSTEM BY SUBSYSTEM

Way, way, *way* back in 1974, I walked out of university into the thick of a recession, having a degree in English, a rusting Chevelle, and little else. English majors rarely get any respect (you have to be an English lieutenant colonel for that) so to keep the blood pumping and the gas tank full I got a job as a Xerox machine repairman.

And I'll never forget the horrible sinking feeling in my guts the first time I saw a Xerox copier with its panels off, merrily making copies. There were gears and drums turning, cams flipping, relays clicking, little claws grabbing a document and dragging it through a maze of harsh green fluorescent lights, past crackling high-voltage corotron wires, under a grimy, grinding developer housing, and ultimately dropping it into a stainless-steel paper chute.

The instructor must have seen the expression on my face. He snorted through his bushy mustache and said, "Hey, Jeff, don't panic! It's only a copier. *Just take it subsystem by subsystem.*"

He was right, of course. It's easy to fall into despair the first time you try to make sense of a programming language. There's five times the complexity of that gross little Model 660 electromechanical copier, and each and every detail must be *exactly* right, or nothing is accomplished but the wholesale tearing of hair.

So do what I did, and take it subsystem by subsystem.

The way to wrestle with complexity

I learned how to fix Xerox machines in record time, and spent the next couple of years wandering around downtown Chicago, keeping the paper pumping. But in learning how to fix Xerox machines, I learned something far more important: How to deal with complexity.

Inside almost any complicated concept (assuming that the concept makes any sense at all) there is something vitally important: *Structure.* The way to understand anything complicated is to develop an eye that sees the structure in complexity, and then to develop a set of selective blinders that allows you to focus in on one element of that structure at a time.

Structure exists in layers, like an onion, and beneath one layer of structure may exist several more, each (in its turn) composed of still more layers.

Start at the very top. Examine one layer at a time. Understand the "big picture" of that one layer only, looking neither higher, to the larger principles, nor smaller, to the component details. Only when you have that layer under your belt do you delve into its component layers, and so on.

You'll find that the Pascal language is wonderfully structured, which makes it easy to grasp once you have your "structure eyes."

Pascal (like Gaul) is divided into three parts

Every Pascal program can be seen as having no more than three separate parts (I won't call them subsystems; parts is parts!) that can be studied separately:

- Constant and data definitions;

- Procedure and function definitions; and

- The main block

Very tiny programs may get away without any procedures or functions (`EatAtJoes.pas` from the last chapter had none) and totally trivial programs may not define any data. All programs, however, must have a main block, and all genuinely useful programs will have all three parts.

Let's take a look at them, one by one.

3.2. THE MAIN BLOCK

The little demo program `EatAtJoes.pas` we compiled in the last chapter has a main block. It's *all* main block, in fact. The main block is the portion of the program delimited between the words **BEGIN** and **END**. Here's the main block from `EatAtJoes.pas`:

```
BEGIN
  ClrScr;
  GotoXY(15,11);
  Writeln('Eat at Joe''s!');
  GotoXY(15,12);
  Writeln('Ten Million Flies Can''t ALL Be Wrong!');
  Readln;
END.
```

BEGIN and **END** are what we call *reserved words*. They have special meanings to the FreePascal compiler and you can't use them for anything else. I'll have more to say about reserved words a little later, when we get into the detailed view of the Pascal language. They are the "framing members" of a Pascal program—the logical 2 X 4s that give a program its structure. They define its shape, and control the way execution flows within a program.

In this book, reserved words will always be printed entirely in upper-case characters, as **BEGIN**, **END**, **WHILE**, **RECORD**, and so on. (Additionally, all program identifiers of *any* kind will be printed in bold in the body text of the book.) In Pascal, character case is not significant for reserved words and other identifiers like the names of variables, so some people write them in lower case. That works; FreePascal considers **BEGIN** and **begin** to be precisely the same. I like to place reserved words in capital letters so that they stand out—it helps you take in the overall structure of a program quickly and easily. Reserved words and variables are *not* the same thing—not even close, in fact—so making them look different is actually *very* useful.

Statements

Between **BEGIN** and **END** are six lines, each of which is a step in the program's execution. When a program begins running, the first step in the main block executes, (here, **ClrScr**) and then the second, and then the third, and so on, until the last step in the main block is executed. Program execution is then finished, and the program stops running.

Each one of those steps is called a *statement*. The single word **ClrScr** is a statement, as is the more complicated line **Writeln('Eat at Joe"s!')**.

Note carefully here that *statements and lines are very different things.* (Beginners often confuse them, with predictably bad results.) A statement may exist all by itself on a single line. A statement may occupy more than one line. More than one statement may exist on a single line. The key is that statements are separated by semicolons. Semicolons don't end lines. They separate statements. This means that you can have a perfectly legal line like the following:

```
ClrScr; GotoXY(15,11);
```

Here there are two statements on one line, with a semicolon after the first to act as a separator. There is a semicolon after the second statement as well, but that semicolon separates the statement **GotoXY(15,11)** from whatever statement begins the next line. Whether you should place multiple statements on the same line is a question of readability, not of program correctness. *The FreePascal compiler does not recognize lines.* In other words, multiple statements on the same source code line compile to the very same object code as do multiple statements each on its own source code line. More on this a little later.

The statements in `EatAtJoes.pas` are simple and easy to dope out by reading them and by watching what the program does. **ClrScr** clears the screen. **GotoXY** moves the cursor to an X,Y position on the screen. Think of your CRT screen as a Cartesian grid like the ones you worked with in high school math. The X (across) value comes first, followed by the Y (down) value. The upper left corner of the screen is the origin, 1,1. Saying **GotoXY(15,11)** moves the cursor 15 positions across, and 11 positions down. **Writeln** writes a line of text to the screen, starting at the cursor position.

There is always a period after the **END** of the main block of a Pascal program. The period indicates that the fat lady has indeed sung, and that the program is over. Control leaves the Pascal program and returns to the operating system.

Compound statements

I'll have a great deal more to say about statements later on in this book. It's important to note that, in a Pascal sense, the whole main block of the program is itself a *compound statement*. In most cases, a compound statement is some number of statements delimited by a **BEGIN** and **END** reserved word. There are a couple of instances where a compound statement may be framed by other reserved words, like **REPEAT** and **UNTIL** rather than **BEGIN** and **END**. We'll deal with these special cases later on.

It might help to characterize the main block by considering it to be a collective statement that indicates the larger, single purpose that the program as a whole was designed to accomplish. Just as a sentence in the English language is a statement made of words followed by a period, so the main block in Pascal is a compound statement made of statements followed by a period. This compound statement summarizes the program's larger purpose, and by reading the main block of a Pascal program first, you should be able to work out the big picture of what the program is supposed to do.

Compound statements appear in many other parts of the Pascal language. You'll be tripping over them wherever you go. When you see one in a program, ask yourself what the unifying purpose of the compound statement is. It'll help you refine your "structure vision" and help you focus on just that one part of the program as a whole.

3.3. VARIABLE DEFINITIONS

`EatAtJoes.pas` has no data at all. It's a dumb billboard, and not especially interesting as a program. Real programs do significant work for us by storing and manipulating data. Another major component of any program, therefore, is a set of definitions that dictate how much data we're using in a program and how we can use it.

Since `EatAtJoes.pas` lacks data, we're going to have to come up with a new program to demonstrate some data-bashing. One begins on the following page. Read it over, and see if you can work out the general sense of what it does based on what you've already learned about the Pascal language. Pascal, for the most part, "talks straight" and tries not to be cryptic. That said, *comments*—explanatory text enclosed in curly brackets—are essential and something that you as a programmer should use to clarify what each individual line of code is intended to do.

```
 1   {------------------------------------------------------------}
 2   {                          Aliens                            }
 3   {                                                            }
 4   {                             by Jeff Duntemann              }
 5   {                             FreePascal V2.2.0              }
 6   {                             Last update 2/8/2008           }
 7   {                                                            }
 8   {     From: FREEPASCAL FROM SQUARE ONE  by Jeff Duntemann    }
 9   {------------------------------------------------------------}
10
11   PROGRAM Aliens;
12
13   USES Crt;                 { For ClrScr, Random, and Randomize }
14
15   CONST
16     MaxLength  = 9;         { The longest name we'll try to generate  }
17     MinLength  = 2;         { The shortest name we'll try to generate }
18     LastLetter = 122;       { Lower-case 'z' in the ASCII symbol set  }
19
20   TYPE
21     NameString = STRING[MaxLength];  { Waste no space in our strings! }
22
23   VAR
24     Printables   : SET OF Char; { Holds the set of printable letters  }
25     I,J          : Integer;     { General-purpose counter variables   }
26     NameLength   : Integer;     { Holds the length of each name }
27     NameChar     : Char;        { Holds a randomly-selected character }
28     NamesWanted  : Integer;     { Holds the number of names we want    }
29     CurrentName  : NameString;  { Holds the name we're working on }
30
31   BEGIN
32     Randomize;                       { Seed the random number generator }
33     Printables := ['A'..'Z','a'..'z'];   { Only printable letters! }
34     ClrScr;
35     Write('How many alien names do you want? (1-10): ');
36     Readln(NamesWanted);        { Answer the question }
37
38     FOR I := 1 TO NamesWanted DO
39       BEGIN
40         CurrentName := '';      { Start with an empty name }
41
42         REPEAT
43           NameLength := Random(MaxLength+1); { Pick length for this name
}
44         UNTIL NameLength >= MinLength;
45
46         FOR J := 1 TO NameLength DO        { Pick a letter: }
47           BEGIN
48             REPEAT  { Keep picking letters until one is printable: }
```

```
49                NameChar := Chr(Random(LastLetter));
50            UNTIL NameChar IN Printables;
51            CurrentName := CurrentName + NameChar; { Add to the name }
52          END;
53        Writeln(CurrentName);  { Finally, display the completed name }
54      END;
55    Readln; { Pause until Enter hit so you can see the names }
56  END.
```

Don't panic!

I mean that. You're not going to be tested on the full details of how the Aliens program works at this point, so don't worry about digesting it whole and in every last little detail. It's a complete Pascal program that even does something interesting. I'll be talking about it for a while in this chapter, explaining most of its workings as I do. So follow along as we go, and don't fret not knowing the details of character sets or the **REPEAT..UNTIL** statement right now. All in good time. Remember, we're going for the big picture here. The details will crystallize out in the chapters to come.

Solving an SF writer's problem

The Aliens program does a job for SF writers like myself who are too lazy to come up with imaginative names for the seventeen-eyed wonders who haunt the starlanes in bad space wars novels. It's a very simple program; if you have the listing typed in already or have obtained the listings archive for this book, I'd suggest that you load, compile, and run `Aliens.pas` right now.

Aliens asks you a question: How many alien names do you want? It then waits for you to type in a number from 1 to 10 and then press Enter. At that point, it will produce the exact number of names you asked for by almost literally pulling letters out of a hat and stringing them together. As each name is completed, Aliens will display that name on the screen. This happens so quickly that all the names will seem to appear instantly.

Zeroing in on data definitions

That's what Aliens does. Now let's take a look at some of its machinery. We're currently focusing on the data definitions part of a Pascal program. Although there is some flexibility about where the data definition part of a program goes, most of the time you'll have to place your definitions at the very beginning of a program.

The data definitions in `Aliens.pas` are shown by themselves on the next page:

```
CONST
  MaxLength  = 9;        { The longest name we'll try to generate  }
  MinLength  = 2;        { The shortest name we'll try to generate }
  LastLetter = 122;      { Lower-case 'z' in the ASCII symbol set  }

TYPE
  NameString = STRING[MaxLength];  { Waste no space in our strings! }

VAR
  Printables    : SET of Char; { Holds the set of printable letters  }
  I,J           : Integer;     { General-purpose counter variables   }
  NameLength    : Integer;     { Holds the length of each name }
  NameChar      : Char;        { Holds a randomly-selected character }
  NamesWanted   : Integer;     { Holds the number of names we want    }
  CurrentName   : NameString;  { Holds the name we're working on }
```

The data definition part of a Pascal program is almost literally a set of blueprints for whatever data the program will be using during its execution. The Pascal compiler reads the definitions and sets up a little reference table for itself that it uses while it converts your source code file to an object code file. This reference table allows the FreePascal compiler to tell you when something you're trying to write as part of a program is bad practice or nonsensical.

Variables as buckets

Variables are defined after the **VAR** reserved word. Think of variables as buckets into which data values may be placed. In variable definitions, you declare the name of a variable followed by its *type*. The name and the type are separated by a colon.

Variables—like buckets—come in a great many shapes and sizes. The type of a variable indicates how large the bucket is and what sorts of stuff you can safely put in it. A plastic water bucket will carry water handily—but don't try to lug molten lead in it. A colander can be thought of as a bucket suitable for carrying meatballs— but don't expect to use it to hold flour or tomato juice without making a mess.

The notion of types in Pascal exists *precisely* to keep you from making certain kinds of messes.

In Aliens, there's a variable called **NameLength**. Its type is **Integer**, which is a signed whole number from -32,768 to 32,767. **NameLength** is thus a bucket for carrying numeric values falling in that range that don't have a decimal part. Similarly, **NameChar** is a **Char** variable, meaning it is intended to hold character values like 'A' or '*'.

Types as blueprints for buckets

So what happens if you try to place a number like 17,234 in a **Char** variable? You can't—the FreePascal compiler won't let you. The two types aren't compatible, so you'll get an error at compile time if you try to load an **Integer** value into a **Char** variable, or vise versa.

Where do types come from? Some of them are built right into FreePascal and are always available. **Integer** and **Char** are two such types, and there are numerous others. On the other hand, Pascal allows you define your own types, and then create variables with those new, programmer-defined types.

`Aliens.pas` contains an example. Notice the statement immediately after the **TYPE** reserved word:

```
NameString = STRING[MaxLength];  { Waste no space in our strings! }
```

This is a *type definition*. It defines a type called **NameString**. This type is a *string* type—meaning that it's designed to contain data in alphanumeric strings of characters like 'I am an American, Chicago-born' or 'THX1138'. FreePascal strings may be from 1 character to 255 characters long and you can create a string type in any size within that range. That's what the type definition statement in Aliens does: It creates a string type with a length given by a constant named **MaxLength**. (We'll get back to **MaxLength** and what it is shortly. For now, just assume it's a number—or look a few lines back to see how it's defined!)

NameString is a very simple type. You can create much, *much* more complex types in FreePascal, as I'll explain later in this book. It's easy to see how a type is in fact a blueprint for making buckets, in that it defines what sort of data some new kind of bucket is meant to contain.

Then, when you actually need a bucket of this new type, you can make one in the **VAR** section:

```
CurrentName  : NameString;  { Holds the name we're working on }
```

This statement gives you a variable in memory that contains string data. The name of the variable is **CurrentName**. The length of the string is given by the **MaxLength** constant.

Remember: A type is *not* a variable and holds no data. It's simply a spec or template that allows you to create variables with a specified size and set of attributes and uses. Use the **TYPE** reserved word to create blueprints for buckets—and then use **VAR** to create the buckets themselves.

Constants as names for data values

So what, then, is **MaxLength**? It's a *constant*, which in Pascal is simply a way of giving a name to a data value. Long before you ever knew what programming was, you were using constants. You learned that the number 3.14159 (approximately) is called "Pi." If you took a little more math, you learned that a constant named "e" (the base of the natural logarithms) was equal to about 2.71828.

The actual values of constants are hard to remember unless you use them all day, every day. I had to look up the value of e just now, even though I know what it is and have used it many times in my life. I just haven't used it often enough to remember it.

Constants serve a very similar purpose in programming: They allow you to give a descriptive name to a data value that might otherwise be hard to recall accurately. Constants have an even more important job, however: They allow you to specify a value *once*, at the top of your program, and then use that value any number of times anywhere else in the program. Later on, if you want to change that value for some reason, you change it in one place only—in the statement where you defined it—rather than having to hunt down dozens or hundreds of uses of the literal value in what might be a very big source code file.

MaxLength is defined in Aliens as being 9, which is a nice maximum length for an unpronounceable name. (Sandron has no trouble pronouncing his name, but we both might stumble on "jhuuTDplb.") We could have defined **MaxLength** as 20, and gotten much longer alien names, or 5, and gotten much shorter ones.

Because we use **MaxLength** only once in Aliens, it's not as obvious how useful constants are as centrally-located definitions for widely-used values. But there is a predefined constant in FreePascal called **Pi**, which you can use anywhere you want instead of the literal value 3.14159. Later on, you can pretend to be an Indiana politician and declare **Pi** equal to 3.0. None of your math will work out, but it's a great insight into the minds (if one could call them that) of politicians.

Be careful not to confuse constants and variables. A variable is a bucket; that is, a container for values. A constant is a value with a name. You can call the numeric value 42 "Ralph" and then put "Ralph" in a bucket. But what the bucket then contains is the number 42. The name "Ralph" is used only to drop the value 42 into the bucket. This will become more clear once you do it a few times.

Loading up your buckets

A variable, once defined, is an empty bucket. It contains no value until you give it one, though it may acquire a value by accident, as I'll explain later on. Accidental

values are not predictable, and can cause many kinds of trouble. Better by far to give every variable a value early on, before it comes upon one by accident!

You can give a variable a value in several ways, but the most straightforward way is through an *assignment statement*. An assignment statement takes a value and assigns that value to a variable. In effect, it takes the value and "loads it into" that variable, as though dropping something into a bucket. Here's a simple assignment statement:

```
Repetitions := 141;
```

What we've done here is taken the numeric value 141 and dropped it into the variable **Repetitions**. The two character sequence **:=** is called the *assignment operator*. When FreePascal sees the assignment operator, it takes whatever is on the right side of the operator and drops it into whatever is on the left side.

Here's an assignment statement from `Aliens.pas`:

```
CurrentName := '';
```

What it does is drop an empty string (that is, a string containing no characters) into the variable **CurrentName**. (A string with something in it would look like this: **'Jeff'**) It may seem odd to think of dropping an empty value into a bucket, so you might consider this a way of emptying the bucket of anything else that might have already been there.

3.4. PROCEDURES AND FUNCTIONS

Compared to `EatAtJoes.pas`, `Aliens.pas` is a considerably larger program. The main block in `Aliens.pas` is 26 lines long. Still manageable—but what happens when you want to write a program that does something genuinely useful? It might take hundreds of lines to write even a fairly simple utility, or easily thousands or tens of thousands of lines to write something like a custom database program to handle sales leads or invoice mail-order sales.

The secret, again, is structure. And without question, the most powerful tools for structuring your programs are *procedures* and the slightly special-purpose procedures called *functions*.

The whole idea in creating a procedure or a function is to gather together a sequence of related statements, and give that sequence a new, descriptive name. Then later on, when you need to execute that sequence of statements, you need only execute the name of the procedure, as though it were a single statement that did everything you wanted done by the group of statements hiding "inside" the procedure.

As easy as brushing your teeth

This sounds hairier than it is. Consider this business of brushing your teeth. You get up in the morning, and as you shake the cobwebs out of one ear or the other, you remind yourself that you can't go to work this time without brushing your teeth.

So brushing your teeth is one single activity. Or is it? Watch yourself as you do it, and take note of the steps involved:

Pick up the toothpaste tube.

Twist off the cap.

Pick up your toothbrush in your other hand.

Squeeze some toothpaste onto your toothbrush.

Sprinkle a little water on the toothbrush.

Put the toothbrush into your mouth.

Repeat:

Work the toothbrush up and down

...until your mind starts to wander.

Rinse off your toothbrush.

Pick up your Flintstones cup.

Fill it with water.

Take a mouthful of water.

Swill it around.

Spit it out.

Taken as the sum of its individual steps, brushing your teeth is a real mouthful. When you actually get down and *do* it, you run faithfully through each of those steps, and if you ever had to tell somebody how to do it, you could. But when you're trying to impose some order on your morning, the whole shebang shows up in your mind under the single descriptive term, "brushing your teeth."

Statements inside statements inside compound statements

Inside the term "brushing your teeth" are thus some number of other terms, in a certain order. You think of the single term "brushing you teeth" to avoid cluttering your mind with a multitude of piddly little details.

A procedure is the same thing, for the same reasons, and works much the same way. You gather together some number of Pascal statements, and then hide them

behind a single identifier of your own choosing. You can execute the new identifier as though it were a single statement, masking the complexity represented by the original sequence of Pascal statements.

Suppose you have these three statements in a Pascal program:

```
DoThis;
DoThat;
DoTOther;
```

Taken together, these three statements accomplish something. Let's call that something "grobbling." (It's a made-up word.) We could say that the following compound statement represents what must be done in order to grobble:

```
BEGIN
  DoThis;
  DoThat;
  DoTOther;
END;
```

We can hide this compound statement behind a single statement, which we'll call **Grobble**, so that any time we need to execute those three statements together, we only have to execute this single statement:

```
Grobble;
```

Doing it is fairly easy. We mostly need to give a name to the compound statement shown above:

```
PROCEDURE Grobble;

BEGIN
  DoThis;
  DoThat;
  DoTOther;
END;
```

What we have here are statements within a compound statement, within a procedure—which itself may be used as a statement.

An adventure in structuring

To make it all click, let's do it, right now, to `Aliens.pas`. This is a slightly advanced exercise, and if you've never written a line of program code before, some of it may puzzle you. Bear with me—and have faith that all will be explained in good time.

One of the things that Aliens has to do is choose a length for any given alien name. It does this by "pulling" random numbers repeatedly until it pulls a random number within a specified range. This range is the range from **MinLength** to **MaxLength**, both of which are defined as constants, and in this case are equivalent to the range 2 through 9.

The code that pulls a random length within those two boundaries is this:

```
REPEAT
  NameLength := Random(MaxLength+1);  { Pick a length for this name }
UNTIL NameLength >= MinLength;
```

FreePascal contains a built-in library function called **Random** that returns a random number *less* than the number contained in the parameter that you pass to it. That's why the value passed to **Random** here is **MaxLength+1**. The parameter is the literal value or variable enclosed in parentheses immediately after the name **Random**. For example, calling the function **Random(9)** will return a random number value between 0 and 8. This value can then be assigned to some other variable (in our case, **NameLength**) for safekeeping. (A *function*, in case you're not yet familiar with the term, is a procedure that returns a value, which may then be assigned to a variable or used in other ways. More on this very shortly.)

The three lines shown above repeatedly get a random number and test it, to make sure it's greater than or equal to **MinLength**. The **Random** function itself guarantees that the value it returns will be no larger than its parameter; that is, the value **MaxLength+1** that the **Random** function holds within its parentheses.

Creating a function

Random numbers are useful things, and it's even more useful to be able to specify a range of values within which a random number is to be pulled. It would be nice to have a procedure of some sort that would pull a random number for us without our having to remember all the precise details of how it was done. We could call the procedure **Pull**, and it would be a sort of number-generating machine. We would pass it a minimum value and a maximum value, and **Pull** would somehow return a value for us that fell within those two bounds.

As I said a little earlier: In Pascal, a function is a procedure that returns a value. I'll have much more to say about functions and how they're used later in this book, but it cooks down to that. To illustrate, suppose we had a numeric variable called **NumberBucket**. We could define a function called **Pull**, and it would be **Pull**'s job to generate a random number. To fill **NumberBucket** with a brand-new random number, we would use this statement:

```
NumberBucket := Pull;
```

Although it may look like one, **Pull** is neither a constant nor a variable. It's actually a compound statement masquerading as a single value. The value is computed within the compound statement, and then returned "through" the name of the function.

With that in mind, let's create the **Pull** function from the three lines in Aliens that pull a random length for alien names:

```
FUNCTION Pull(MinValue,MaxValue : Integer) : Integer;

VAR I : Integer;

BEGIN
  REPEAT
    I := Random(MaxValue+1);  { Pick a length for this name }
  UNTIL I >= MinValue;
  Pull := I;
END;
```

Things have changed a little—but for a reason, as I'll explain. The central portion of **Pull** does the same thing that the three lines we lifted from the Aliens program did. What we've mostly added is framework; a body for the lines to exist in.

But we've also added a strong measure of generality. The lines that pull a random name length within Aliens can be used *only* to pull a random name length. The new **Pull** function can be used to pull random numbers within a specified range for any reason. We could use it just as easily in a dice game as in an alien name generator—and that's a big, *big* advantage.

We can use **Pull** in Aliens.pas. Just insert the definition for **Pull** in the program file just before the beginning of the main block, and then replace the three lines that pull a random name length with the following single line:

```
NameLength := Pull(MinLength,MaxLength);
```

The two items **MinLength** and **MaxLength** are called *parameters*. They're special-purpose variables belonging to the function. They work as pipelines, allowing you to drop values into the function. Drop a "7" value into **MaxLength**, and the **Pull** function's internal machinery will receive the value 6 (1 less than **MaxLength**) as the maximum allowable value for the random number it's been told to generate.

The altered copy of Aliens.pas (let's call it Aliens2.pas) is shown on the next page, with the **Pull** function replacing the three lines used to pull random numbers in the original Aliens.pas.

```
 1   {----------------------------------------------------------------}
 2   {                              Aliens2                            }
 3   {                                                                 }
 4   {                               by Jeff Duntemann                 }
 5   {                               FreePascal V2.2.0                 }
 6   {                               Last update 2/8/2008              }
 7   {                                                                 }
 8   {    From: FREEPASCAL FROM SQUARE ONE   by Jeff Duntemann         }
 9   {----------------------------------------------------------------}
10
11
12   PROGRAM Aliens2;
13
14   USES Crt;               { For ClrScr, Random, and Randomize }
15
16   CONST
17     MaxLength  = 9;    { The longest name we'll try to generate  }
18     MinLength  = 2;    { The shortest name we'll try to generate }
19     LastLetter = 122;  { Lower-case 'z' in the ASCII symbol set  }
20
21   TYPE
22     NameString = STRING[MaxLength];  { Waste no space in our strings! }
23
24   VAR
25     Printables   : SET OF Char; { Holds the set of printable letters  }
26     I,J          : Integer;     { General-purpose counter variables   }
27     NameLength   : Integer;     { Holds the length of each name }
28     NameChar     : Char;        { Holds a randomly-selected character }
29     NamesWanted  : Integer;     { Holds the number of names we want    }
30     CurrentName  : NameString;  { Holds the name we're working on }
31
32   FUNCTION Pull(MinValue,MaxValue : Integer) : Integer;
33
34   VAR I : Integer;
35
36   BEGIN
37     REPEAT
38       I := Random(MaxValue+1);  { Pick a length for this name }
39     UNTIL I >= MinValue;
40     Pull := I;
41   END;
42
43
44   BEGIN
45     Printables := ['A'..'Z','a'..'z'];
46     Randomize;                      { Seed the random number generator  }
47     ClrScr;
48
49     Write('How many alien names do you want? (1-10): ');
```

```
50    Readln(NamesWanted);                  { Answer the question }
51
52    FOR I := 1 TO NamesWanted DO
53      BEGIN
54        CurrentName := '';   { Start with an empty name }
55
56        NameLength := Pull(MinLength,MaxLength); { Pull random length }
57
58        FOR J := 1 TO NameLength DO          { Pick a letter: }
59          BEGIN
60            REPEAT   { Keep picking letters until one is printable: }
61              NameChar := Chr(Random(LastLetter));
62            UNTIL NameChar IN Printables;
63            CurrentName := CurrentName + NameChar; { Add to the name }
64          END;
65
66        Writeln(CurrentName);    { Finally, print the completed name }
67      END;
68    Readln;   { Pause until Enter hit so you can see the names }
69  END.
```

Hiding complexity

The details of having to pull a number again and again until one appears that falls within a specified range are masked now. We only see the "front door" of the random number factory. The machinery that actually builds the random numbers is hidden away behind the door somewhere. And that's good, because most of the time we really don't care *how* random numbers are made; we only care that they *do* get made, and made according to our specifications.

It's true that creating the **Pull** function added a few lines to the program. Later on, we'll see how we can remove **Pull** from `Aliens2.pas` and place it in a library of functions and procedures called a *unit*. This library is available to any of your programs that need it. If ten of your programs need a random-number puller, you can give a random number puller to all ten of them—and yet have only one copy of **Pull**'s ten lines of code. If you yank enough general-purpose procedures and functions out of your programs into libraries, you can actually cut the source code bulk of your programs considerably.

Organizing programs with procedures

Hiding details is the fundamental purpose of procedures and functions. The little example given here is not an especially good one, since there aren't a lot of details to be hidden in a random-number puller.

Now, consider a simple accounting program. A good one (even a simple one, as accounting programs go) might have 10,000 lines of code. You could write all 10,000 lines of code in one enormous main block. But how would you read and understand those 10,000 lines of code? You'd probably have to do what was done in the ancient days of programming, and literally cut a long program listing up into chunks with a scissors, and only look at the chunk you needed to concentrate on at the moment.

Even *War and Peace* is divided into chapters. Procedures are often used as "chapters" in a larger program. In our accounting program example, you would have several accounting tasks like Payroll, Accounts Payable, Accounts Receivable, General Ledger, and so on. It's possible to make each of these accounting tasks a procedure:

```
PROCEDURE AccountsPayable;

BEGIN
  <about 2000 lines>
END;


PROCEDURE AccountsReceivable;

BEGIN
  <about 3000 lines>
END;


PROCEDURE Payroll;

BEGIN
  <about 2000 lines>
END;


PROCEDURE GeneralLedger;

BEGIN
  <about 2200 lines>
END;
```

Now at least you have a fighting chance. When you need to work on the payroll portion of your accounting program, you can print out the **Payroll** procedure and ignore the rest. All the "payroll-ness" of the accounting program is concentrated right there in one procedure, so you don't have to go searching for payroll details across the entire program. Better still, details that you *don't* have to pay attention to right now remain hidden, inside their respective procedures.

With your payroll blinders on, you'll have a much easier time focusing on the payroll part of the program. The main block becomes quite small then, and is mostly a little menu manager that lets you choose which of the four big procedures you want to run.

Procedures within procedures

If you're perceptive, you may have noticed something else about procedures and functions: They look like little programs. They *are* little programs, in fact. Procedures and functions share the same general structure with the "big" Pascal programs that contain them. This resemblance between Pascal programs and procedures and functions is reflected in a generic term that encompasses both procedures and functions: *subprograms.*

If Pascal programs are divided into three parts, then so are procedures and functions: They too have constant and data definitions, procedure and function definitions, and a main block. You may have noticed that the **Pull** function had a variable definition in it, for an integer variable named **I**. The variable **I** is what we call a *local variable,* by which we mean one that belongs inside **Pull** and never ventures outside **Pull**'s "city limits." Only **Pull** can see or use the variable **I**. Procedures and functions can define any number of local variables and constants for their use, just as programs can.

But most interestingly, procedures and functions can define and contain their own local procedures and functions. Just as **Pull** was inside and thus local to Aliens2. PAS, you could place a procedure or a function (or several of them) inside **Pull** as well. These local procedures and functions would be for **Pull**'s private use only; no subprogram outside of **Pull** could make use of them.

This nesting can go on and on almost without limit: You can place subprograms within subprograms within subprograms, like those little barrel-shaped Russian dolls that fit one inside the other.

3.5. PROGRAM STRUCTURE RECAP

This chapter has been an overview of program structure, to get you started on your road to Pascal mastery in the right frame of mind, and with a general method for finding your way. I've touched on a great many topics without going into any of them especially deeply. In the rest of this book, I'll begin describing the Pascal language in detail.

But before getting into a hurricane of details, I wanted to lay the big picture out in front of you. Let's recap the big picture.

Pascal programs are divided into three parts

These three parts are data definitions, procedure and function (subprogram) definitions, and the main block.

Data definitions include constant definitions, type definitions, and variable definitions. Constant definitions give descriptive names to unchanging values. The value 3.14159, for example, is often made a constant with the name **Pi**. Type definitions define a particular kind of data, and act as blueprints for creating variables of that kind of data. A type definition defines how large a data type is, and what sorts of ways it can be used. A variable definition creates a little storage bucket somewhere for a particular type of data. Think of type definitions as blueprints for data buckets, and variable definitions as the way the buckets themselves (the variables) are actually made. Constants, in turn, are one (but only one) method by which those buckets may be filled with data.

The main block is a compound statement, which is a sequence of Pascal statements between the reserved words **BEGIN** and **END**.

Procedures and functions allow groups of Pascal statements to be grouped together and "hidden" behind a single descriptive name. Using that single name has the same effect as using each of the individual statements within the procedure or function. Functions are specialized procedures that return values that may be assigned to variables in assignment statements. Apart from that, they are identical to procedures.

Procedures and functions are the primary means of masking complexity in a Pascal program. The details of a larger program task may be hidden within a procedure or function, so that your attention to that larger function will not be distracted by the details. When you need to be concerned with the details, you can examine the definition of the procedure or function to see how it operates internally.

When you need to see the details, you can. When you don't, you won't. Much of structuring a program is the artful hiding of details.

Procedures and functions (subprograms) are miniature programs

The essential structure of a Pascal program is echoed in the structure of subprograms, which is in fact why procedures and functions are called subprograms.
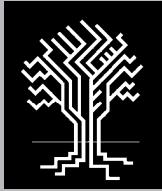
Procedures and functions have data definitions and a main block, and furthermore, they can contain their own local (that is, private) procedures and functions. This nesting of procedures within procedures has no explicit limits, and may continue as far as needed. (There is a "too far," and moderation is a virtue in Pascal programming as anywhere else.)

Programs are organized by dividing them up into subprograms

A single monstrous program can be made conceptually manageable by dividing it up into subprograms. Large subprograms can themselves be made manageable by dividing them internally into separate, smaller subprograms. By creating a program in layers in this way, you can focus your attention only on the layer that you need to see, without distraction either from the big picture at a higher level, or from details at a lower level.

Take it one subsystem at a time!

Most of all, I want you to remember the lesson of the technical trainer back at Xerox in 1974: *Take it one subsystem at a time.* Being boggled gets you nowhere. Develop your structure-eyes, and learn to look selectively at a program to see the structure in it, and then either work to understand that structure (if the program was written by someone else) or to create that structure (if the program doesn't exist yet) using the structuring tools available in FreePascal.

PART II:

INSTALLING, LEARNING, AND USING FREEPASCAL AND LAZARUS

I should note here that "using" Lazarus in this book does *not* include using its GUI builder. Explaining that will require a full treatment of object-oriented programming and software components, which is outside the scope of this introductory book.

Chapter 4.
Installing FreePascal and Lazarus

O ne of the (few) downsides to free and open-source software is that you can't cover installation by saying, "Order the product, open the box, and follow the manufacturer's instructions." It's a little more complex than that, especially when the product (like FreePascal) can be run on a surprising number of different operating systems and CPU types. So in this chapter, I'll explain how to find the downloadable install suites for FreePascal, how to be sure you have the right one, and then how to install it for Windows and Linux.

4.1. Platforms and targets

Most of us are used to either buying a software product in a box at a local store, or else ordering a product (or at least a product install CD) online. FreePascal is different from a lot of software products in that it exists for a number of different platforms. A *platform* is an operating system running on a particular type of hardware. For example, Microsoft Windows running on 32-bit Intel hardware like the Pentium is a platform. (It's probably the commonest platform in the world, and often called "Wintel.") Ubuntu Linux running on a 32-bit Intel CPU is considered a separate platform. Similarly, the PowerPC CPU is the hardware in many older Apple Mac machines. The OS X operating system running on a PowerPC CPU is considered still another platform. And OS X running on an Intel CPU is considered yet another platform.

This is all made trickier by the fact that some operating systems run on multiple hardware types, and most CPUs can run multiple operating system. So you must be aware of both the CPU type in your machine, and the operating system you intend to install FreePascal under.

At this writing (June 2019) the list of installable binary files for the various supported platforms is provided on Sourceforge:

https://sourceforge.net/projects/freepascal/files/

4.2. THE RELATIONSHIP BETWEEN FREEPASCAL AND LAZARUS

As I explained earlier in this book, FreePascal is a compiler, and Lazarus is an integrated development environment (IDE). It's true that Lazarus is more than just an IDE—it contains a world-class "GUI builder," as I'll explain in the next chapter—but while you're first learning FreePascal, the IDE features of Lazarus are all you're going to need. (I intend to cover the GUI builder portion in its own book.)

Just keep this in mind: FreePascal and Lazarus are two separate software projects with separate teams of programmers writing, testing, and debugging them. FreePascal and Lazarus "cooperate" in many ways, at several levels, so it makes sense to use them together. The relationship between the two can be summed up this way:

- Lazarus is a program written in FreePascal.

- Lazarus executes the FreePascal compiler "behind the scenes" to generate executable code from source code managed from within Lazarus.

Basically, FreePascal can function without Lazarus, but Lazarus cannot function without FreePascal. You can edit and manage FreePascal source code from other editors if you want (including the text-mode IDE included with the compiler) but none have the power of Lazarus, simply because Lazarus "knows" the Pascal language, and knows it better than any other free program I've yet tested. However, you cannot use other compilers from within Lazarus. (Yet; it's been discussed and may happen someday.)

Stable versions vs. development versions

Because FreePascal and Lazarus are two separate software projects, they each have a separate and independent version number. At this writing (November 2019) FreePascal is at version 3.0.4. Lazarus is at version 2.0.6. These are the "stable" releases; that is, releases that have been tested extensively and are known to be mostly functional and largely bug-free. (No software is ever entirely bug-free!)

Both products are open-source, and are maintained by volunteers scattered all over the world. Because FreePascal and Lazarus are not developed in secret at a single large software house, intermediate releases are publicly available to anyone who wants them. These are "development" releases, issued by the programming teams for testing and debugging purposes, and are really intended for installation by programmers interested in working on the FreePascal and Lazarus programs themselves. Development releases are not guaranteed to be stable. They may contain partially completed features and outright bugs, but that's what they're out there for: to broaden the base of testers who can bring bugs to light and fix them.

Until you're well-versed in compiler internals, *stick to the stable releases.*

The stable releases of both products may be obtained from the FreePascal and Lazarus Web sites. I generally install each new stable version as it appears, because both products (especially Lazarus) are evolving quickly. If you register as a user on the Lazarus Web forum, you'll be sent an email when a new release is ready to download. Or you can just check the Lazarus announcements page every so often:

https://www.lazarus-ide.org/

What comes with what

Because Lazarus requires FreePascal to do its work, the downloadable installers for Lazarus also contain the FreePascal compiler, and when you install Lazarus you install both. However, the release schedules for FreePascal and Lazarus are not synchronized. The downloadable installer for the newest stable release of Lazarus does not necessarily contain the newest stable release of FreePascal.

This is almost never a problem, especially while you're still learning the Pascal language. The features of Pascal that I'll be teaching in this book haven't changed much in a while and probably won't change much in the near future. In general, you're better off with the current stable version of Lazarus irrespective of which stable version of FreePascal is installed with it.

The text-mode IDE

By default, when you install FreePascal by itself (without Lazarus) the FreePascal install wizard will install a program called `fp.exe`, which is the text-mode IDE that runs in a console window. You can see what it looks like by turning back to Figure 2.3. There's a quirk here: The Lazarus installer installs FreePascal, but it does not install `fp.exe`. The logic there is that if you're going to install Lazarus, you're probably not going to use the text-mode IDE.

I deliberately chose not to use the text-mode IDE for the examples in this book because I've had a some bad experience with it, especially when run in a console window under Windows. On other platforms it has worked well for me. However, if `fp.exe` crashes or displays poorly, I don't have a great deal of advice other than to use another standalone editor and invoke the compiler from the command line. Lazarus is a great deal more stable, especially under Windows, and if you're just learning the language and are working in Windows or Linux under GNOME or KDE, it's probably the easiest way to proceed. Furthermore, if you intend to do full-bore object-oriented GUI app programming later on, you might as well start learning Lazarus now.

FreePascal's Documentation

FreePascal has a substantial set of manuals distributed as electronic files. The manuals may be downloaded as individual files, or they are available online in HTML format, where you can read them with an ordinary Web browser.

The available manuals for FreePascal are these:

- The *FreePascal User's Guide* (user.pdf) is a concise description of the Pascal language as implemented by the FreePascal compiler, including "railroad" syntax diagrams. The descriptions are terse and should not be considered tutorials in any sense of the word.

- The *FreePascal Programmer's Guide* (prog.pdf) explains programming issues that go beyond the fundamentals of Standard Pascal. This includes language extensions, compiler directives, low-level issues like assembly language interface and external calling conventions.

- The *FreePascal Language Reference Guide* (ref.pdf) explains how to install and configure the compiler and how to invoke it. This includes compiler modes, compiler error messages, debugging using gdb, and a list of the standard units installed with the compiler. (The units are not described in detail; for that see the *Runtime Library Reference Guide*.) A large part of the file explains how to use FreePascal's text-mode IDE fp.exe, which I won't be discussing in detail in this book.

- The *Runtime Library (RTL) Units Reference Manual* (rtl.pdf) describes all of FreePascal's standard units in detail, procedure by procedure. If you want to know how to call the **GotoXY** procedure, this is where to look, keeping in mind that the descriptions of individual procedures and functions are necessarily brief and technical. Note: The file is *2,034* pages long.

- The *Free Component Library Units Reference Manual* (fcl.pdf) is a reference for the FCL object-oriented library that is used for writing component-based programs in Pascal. (Note that the FCL is not the same as the Lazarus Component Library, or LCL.) The FCL is an advanced topic that I won't be covering in this book, so this file won't be of much use to you in your first steps as a Pascal programmer.

- The *Compiler Switch Summary Chart* (chart.pdf) is a two-page document with a summary of global compiler switches on one page, and local compiler switches on the other. Don't worry if this isn't meaningful right now; most compiler switches aren't used in tutorial examples.

There are HTML versions of all documentation volumes listed here except for the compiler switch summary chart.

All of the manuals mentioned above may be downloaded in either HTML or PDF format from the FreePascal Web site:

https://www.freepascal.org/docs.var

The documentation for the version current at this writing (June 2019; v3.0.4) may also be downloaded from SourceForge. The URL below will probably change when new releases happen and the version number changes. Searching for "freepascal documentation" will find the current documentation without difficulty.

https://sourceforge.net/projects/freepascal/files/Documentation/3.0.4

Paper Documentation

You can view the documentation in a Web browser, or download the doc files from the Web, but having a paper copy of the manuals can be useful. All the documentation files are available in PDF format, and if you've got a printer capable of duplexing, it can be useful to print the files and place them in 3-ring or duo-tang binders. The page size is A4, widely used in Europe, but the pages will print well if scaled to fit American letter-size sheets.

Not all of it needs to be printed. The *Runtime Library Reference Guide* is an immense file, but a lot of the material in the libraries is relatively arcane and won't be anything you'll need to understand as a beginner. For your initial learning exercises, make sure you have a PDF reader with a search function—and learn how to use it! I had a local print shop print several documentation volumes for me with a spiral binding that lies flat on the desktop or propped on a copy frame beside your monitor. Be aware that some print shops may be hesitant to print copies of files that you obviously didn't create. In such cases it may help to point out that these are the manuals for free software, and that there are no copyright notices in the files.

4.3 Installing FreePascal and Lazarus for Windows

How you install FreePascal depends heavily on what you intend to use for your working IDE. FreePascal's text-mode IDE `fp.exe` is installed by default when you install FreePascal by itself. When you install Lazarus, FreePascal is installed automatically, because Lazarus uses FreePascal to compile all of its code in the background. However, when you install Lazarus, you will not get `fp.exe` as part of the install.

Installing FreePascal by itself

If you're not interested in using Lazarus as your IDE, you can install FreePascal by itself. Select a version from the Downloads page for FreePascal:

https://www.freepascal.org/download.var

If you're reading this book a long time after I posted the file (currently, June 2019) the latest stable version of FreePascal may not be the one I call out here. The current version number will be shown on the page. Links to all the available platforms will be there.

Clicking on a version will take you to a brief page allowing you to select a download mirror. (I always use SourceForge.) Once the FreePascal Web site hands you off to SourceForge, look first for a big green button reading "Download Latest Version!" Click the button, and the installer file will begin coming down. If for some reason the download doesn't begin immediately, click on the direct link just under the title reading, "Your FreePascal Compiler download will start shortly…"

This book shows examples running in a console window under Windows. The installer for Windows is an ordinary Windows executable file. Close all open Windows applications and run it, either from the Run window or by navigating to the installer file and double-clicking on it. The installer will open a conventional Windows install wizard. All of the wizard's fields provide useful default values, so you can accept all the default values in each one of the wizard's dialogs. When the wizard has completed its series of dialogs and terminated, you'll see a new icon on your desktop, for the FreePascal IDE.

Watch out for spaces in pathnames!

By default, the FPC wizard installs FreePascal at C:\FPC. This may seem strange at first. Most people are used to installing Windows apps in the Program Files directory tree, but remember that even though FreePascal can be run from Windows, FreePascal is *not* a Windows GUI app. It's a console app. I'd recommend leaving FreePascal at C:\FPC, or perhaps D:\FPC. The reason for this is a little odd: FreePascal may not deal perfectly with spaces in pathnames. The safest way is to install FreePascal in the root directory of one of your fixed (not removable) hard drives.

This caution against spaces in pathnames applies more generally, to any circumstance where FreePascal has to deal with a directory path; say, when you save projects to a directory. Use dashes or internal capitalization if you must, but *don't create paths containing space characters if the path will have to be used by FreePascal*. There may be technical reasons why this is the case, but it's something I consider a bug and hope will be fixed someday.

Downloads

You can download Lazarus 2.0.6 which is accompanied by FPC 3.0.4 from this page.

Lazarus is cross platform and supported on various platforms. Choose your platform to go to the corresponding page:

Windows

- Windows (32 and 64 Bits) Direct download
- Windows (32 Bits) Add ons
- Windows (64 Bits)
- Windows (64 Bits) Add ons

Linux

DEB Releases
- Lazarus Linux i386 DEB (32 Bits)
- Lazarus Linux amd64 DEB (64 Bits)

RPM Releases
- Lazarus Linux i386 RPM (32 Bits)
- Lazarus Linux x86_64 RPM (64 Bits)

Mac OS X

- Lazarus Mac OS X i386 (32 Bits)

Sources

- Lazarus Zip - GZip

MD5 and SHA Checksums

- See this page for md5 and sha-1 checksums of the official downloads

Other Downloads

For other downloads, check: The SourceForge Project page

Mirrors

It is recommended to use Sourceforge.net for all the downloads.

However, if you cannot access sourceforge.net, then you can use mirror links:

- ftp://freepascal.dfmk.hu/pub/lazarus/releases/
- http://michael-ep3.physik.uni-halle.de/Lazarus/releases/
- http://mirrors.iwi.me/lazarus/

Figure 4.1. The Lazarus downloads page.

Downloading Lazarus

If you install the Lazarus environment, FreePascal comes in right along with it. *You do not need to install FreePascal before you install Lazarus.* As with FreePascal, there is a Windows installer file for Lazarus. In short, you download the installer and run it, accepting or tweaking the values presented by the install wizard as appropriate. Note well that the Lazarus installer does *not* install FreePascal at C:\FPC, but instead as a subdirectory beneath the Lazarus install directory.

The Lazarus Web site may be found here:

http://www.lazarus-ide.org/

This is the Lazarus "home page" and a good starting point when looking for the latest on Lazarus. There's a link to the Windows installer, and another link to the Downloads section at the right margin, linked from the word "Other," which will take you to the Lazarus download page. (See Figure 4.1.) Click the link for the version you want, which will take you to the file's page on Sourceforge.

While you're there, you might want to download the Lazarus documentation. The Lazarus manuals do not exist as PDF files at this writing (June 2019) but rather as .chm files. Given that .chm files are compiled HTML, you can open them with (almost) any Web browser. The documentation folder may be found here:

https://sourceforge.net/projects/lazarus/files/Lazarus Documentation/

I generally read them online, starting at this link:

http://wiki.freepascal.org/Lazarus_Documentation

Installing Lazarus and (along with it) FreePascal

As I mentioned earlier, installing Lazarus under Windows is easy. You run the .exe installer, and answer the wizard's questions. You can specify the language used by the product and where it is to be installed. The caution I mentioned earlier about spaces in pathnames applies to Lazarus as well as Freepascal, so I recommend letting it install at the default location, which is C:\lazarus. You're allowed to specify the file associations recorded by Windows for Lazarus, and unless you have another compiler or IDE associated with the .pas extension, accept all the defaults.

When the wizard runs its course and exits, you'll have a new directory, C:\lazarus by default, and (if you checked the box specifying the creation of a desktop icon) the blue Lazarus icon on your Windows desktop. FreePascal will be present in its own directory under C:\lazarus, but it will not get a separate desktop icon, because it is not a Windows GUI app.

To run Lazarus, just double-click the desktop icon, as with any Windows app.

Note that when you install FreePascal as part of the Lazarus installation, the text-mode IDE fp.exe will not be installed. If you want to experiment with fp.exe after installing Lazarus, I suggest installing FreePascal separately. There's nothing wrong or hazardous about having two copies of FreePascal installed on a single hard drive, though the second copy will cost you about 150 MB in disk space.

Installing FreePascal separately under Windows will create a desktop icon for the text-mode IDE, and you run the IDE by double-clicking on the icon. Keep in mind that the operation of the text-mode IDE can be erratic under Windows, so if it crashes or looks peculiar, there may not be much you can do about it. It's much better to install and use Lazarus, which has a far superior IDE for Pascal.

Again, there is much more to Lazarus than you need in order to learn the fundamentals of the Pascal language, and much more than I'll be covering in this first book. I recommend configuring Lazarus to hide the IDE windows that you won't need, to keep the clutter down and make the product easier to grasp. I'll explain how to do this in Chapter 5.

4.4 INSTALLING FREEPASCAL AND LAZARUS UNDER LINUX

I'll have to be honest here: Installing Lazarus under some distributions of Linux is not as easy as it is under Windows. FreePascal and Lazarus are available from Open Source software repositories, and most Linux desktop distributions have mechanisms built into their graphical user interfaces to download and install all necessary files pretty much automatically. (For example, the Ubuntu Software Store.)

This sounds great, but there can be problems. There are several different package formats in common use for Linux, and the Lazarus package for each one is different, with each maintained by a different team. So if you download and install Lazarus and FreePascal through a distro package manager, you may not get the same versions from all repositories in all formats.

At this writing, packages are available in the RPM and Debian formats. Linux distributions that don't support either of these formats (and I can't think of any off the top of my head) will need to build Lazarus and FreePascal from source. Explaining that is beyond the scope of this book, and may change from version to version as the software evolves. Long-time Linux programmers will probably know how to do it, and Linux newcomers who want to learn should sniff around the online forums, particularly the Lazarus forum on the FreePascal Website:

http://www.lazarus.freepascal.org/index.php?action=forum

Installing under Fedora 13

Obtaining and installing Lazarus and FreePascal under Fedora 13 is about as easy as it gets. Fedora uses the RPM package format, and the FreePascal/Lazarus RPM is well maintained. Installing it is best done with the YUM package manager. If you're logged in as root, open a console window and enter this command:

```
yum install lazarus
```

If you don't work as root (and that's generally a wise policy) you'll need to use su to launch yum:

```
su -c 'yum install lazarus'
```

You'll be asked for your root password. After that, yum takes off and begins downloading packages and checking dependencies. Once it determines what it needs to download, it'll check with you:

```
Total download size: 85 M
```

```
Is this ok [y/n]:
```

Type "y" to continue. The number of bytes to download will vary based on the version and what you already have installed. 80-100 MB is typical. How long it takes to come down will vary depending on the quality of your broadband connection. On my cable modem system, the whole yum process took about 25 minutes.

Installing under OpenSuse

The OpenSuSE Web site has a marvelous technology called the OpenSuSE Build Service (OBS), which makes installing Lazarus on OpenSuSE extremely easy. The search/install page can be found at this URL:

```
http://software.opensuse.org/search
```

The OBS is a platform for distribution and installation of open source software. It uses the YaST installation/configuration manager, and provides "1-click install" buttons on the search page. Calling it "1-click" is a bit of an exaggeration, but the big win is that you don't have to type in the names of any packages or repositories, enter any scripts, or do any manual unpacking or building. One click kicks off the process, which launches an installation wizard. Run through the wizard, answer its questions (which does require a few additional clicks) and YaST will begin installing the selected package.

Here's a step-by-step:

Get Software from the openSUSE Build Service

lazarus

openSUSE 11.2   Search
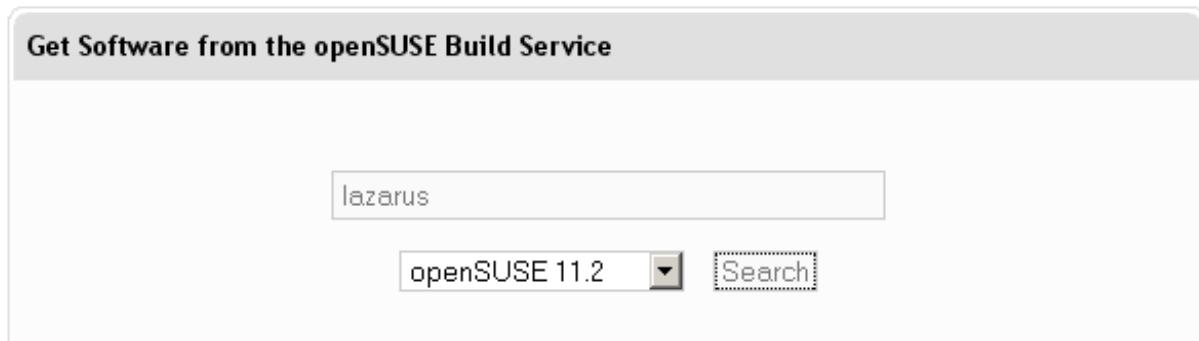
Figure 4.2. The OpenSuSE Build Service search page.

1. Go to the OBS search page. (See Figure 4.2.)

2. Type "lazarus" into the search field.

3. Select your distribution from the drop-down list box.

4. Click the Search button.

5. Several search hits will appear. Any of them should work (I haven't tried them all) but in my view the best is likely to come from the devel: languages repository. The repository is shown in the upper right corner of each search hit summary.

6. Find the one from devel:languages. Click on the "1-Click Install" button.

7. An installation wizard will appear. Run through the wizard, changing anything that needs changing. I was able to keep all the default options.

8. When you've reached the last pane of the wizard, click Finish. YaST will then begin downloading files, checking dependencies, and doing whatever might be necessary to install Lazarus and FreePascal on your OpenSuSE system. This will take a while (perhaps 20-25 minutes at most, assuming a broadband Internet connection) but when it's done, you've got it!

Installing under Ubuntu

As easy as it is to use, Ubuntu's Software Store (in place since Karmic Koala 9.10) has some issues with respect to Lazarus. Ubuntu and most other Debian-based distros use the Debian Package Manager format for software installation. Debian Packages for a given product install easily through the Store, but they have to be correctly built and maintained as new releases of the product appear. The Debian package

for Lazarus has not been well-maintained over the last several years. The version of Lazarus and FPC installed are not the latest, and (peculiarly) the FreePascal source is not included in the package, even though Lazarus requires it. So if you install Lazarus from one of the Debian packages downloadable at this writing (late 2017), Lazarus will complain when you launch it that the FreePascal source code is not available, and that some of its features (not stated) will not work completely.

This is not the fault of the people who wrote and maintain FreePascal and Lazarus, because they do not have complete control over how the Debian *package* for the product is maintained, which is a separate issue from maintaining the product itself.

So even if you find FreePascal and Lazarus in the Store, don't install from it. Instead, follow the instructions on this page in the FreePascal wiki:

http://wiki.freepascal.org/Lazarus_release_version_for_Ubuntu

The wiki page will provide a "canned" script that you can save and run to execute all the server commands that the install requires. It also provides instructions step by step, and you should read those simply to understand what the script is about and how it works.

There are some specific instructions to be followed if you intend to run Lazarus under Canonical's Unity interface. I don't like Unity and have not tried this, but I'm guessing that it works just fine.

Installing on the Raspberry Pi board

I was surprised but delighted to find that the low-cost Raspberry Pi computer board can run Lazarus and FreePascal. The early boards were severely short of RAM, but V3 has a full gigabyte, and the V4 board has versions with up to 4 GB. If you're going to program on the RPi, I'd powerfully suggest buying the V4 board with 4 GB.

Unfortunately, the installation methods you'll find posted online—even on the Lazarus Web site itself—sometimes don't work, and sometimes install an older version of the product. The best install instructions I've seen can be found at this Web site:

https://www.getlazarus.org/setup/

As with a lot of software under active and rapid development, things can change, and the installation script available at getlazarus.org may not work on future versions of the Raspberry Pi board or of FreePascal and Lazarus. You may have to dig a little deeper on Google or ask questions on the Lazarus forums, as I describe on the next page.

4.5. Miscellaneous Install Notes

Software installation is an untidy business, because both software and the machinery that installs it change over time. There was a time when we did not have either the Ubuntu Software Store nor the OpenSuSE Build Service, and the descriptions in this chapter would have been different had this book appeared in that era. So in the last section of this chapter, I'll include whatever odd notes I have on installing Lazarus and Freepascal that didn't fit any of the earlier sections.

Obtaining the text-mode IDE for Ubuntu with apt-get

As with Windows, when you install Lazarus for Linux, the FreePascal text mode IDE is not installed with it . Nor does the IDE have a record in the Ubuntu Software Store. To install the IDE, you have to open a terminal window and manually request the IDE program through apt-get, using this command:

```
sudo apt-get fp-ide
```

After you enter your account admin password, the apt-get mechanism will download, unpack, and install the IDE for you. Once installed, you run the IDE by opening a terminal window and typing the command fp. The IDE that appears runs in text mode, but it uses color.

Now, depending on what version of FreePascal and Linux you're using (and recognizing that bugs are fixed regularly in the Open Source world) you may see a peculiar error message when you run FreePascal's text-mode IDE, warning against (possible) problems with Debian bug # 412927. This bug involves mouse and mouse-wheel support under Debian-based distros, but in my own work in Ubuntu I have had no trouble at all. By the time you read this, that bug may have been addressed and you may not see the message, but if you do, there's no reason to panic. Try the text-mode IDE and see if mouse support works. If not, you also have the option of creating FreePascal text-mode programs in the Lazarus IDE. That's the approach I will be assuming in this book.

Getting installation help on the Lazarus forums

Especially under Linux, installing Lazarus can be a complicated business. Although debugging the process and seeing it through is good discipline and an excellent learning experience, you may at some point get stuck. Don't despair: Others have been there before you, and there is a lot of good advice to be had on the Lazarus Web forums:

```
http://www.lazarus.freepascal.org/index.php?action=forum
```

You have to register to post questions. Don't hesitate to join and register; it's worthwhile. You will not get emails from them trying to sell you things, as so many modern online forums do.

Do remember that others may have solved the same problems you're having, possibly a long time ago. Read over any posts regarding installation before posting what may be a redundant question. In fact, plan to spend a day scanning the many threads to see what's there. You will learn a great deal, especially as a newcomer.

"Give me a lever long enough, and a place to stand, and I will move the Earth." In perhaps his best-known statement, Archimedes was speaking literally about the power of mechanical levers, but behind his words there is a larger truth about work in general: To get something done, you need a place to work, with access to tools. My radio bench down in my workshop is set up that way: A large, flat space to lay ailing transmitters down, and a shelf above where my oscilloscope, VTVM, frequency counter, signal generator, and dip meter are within easy reach.

Much of the initial success of Turbo Pascal forty-odd years ago was grounded in that simple but compelling truth. For the first time, a compiler vendor had assembled the most important tools of software development and put them together in an intuitive fashion so that the various tasks involved in creating software flowed easily and quickly from one step to the next.

5.1. MEET LAZARUS, FREEPASCAL'S RAD ENVIRONMENT

Turbo Pascal versions 1 through 3 were pretty effective in this way. Later versions were even moreso, reaching a sort of optimum for text-mode DOS programming with Borland Pascal 7 in 1991. Back in early 1995, Borland shook the development world with the first release of Delphi. Delphi was more than just an IDE for Pascal, and certainly more than the Borland Pascal 7 IDE rewritten to run under Windows. Three major things set Delphi apart from earlier versions of Turbo/Borland Pascal:

- Delphi contained a whole new mechanism for creating user interfaces by dragging and dropping components like buttons and scroll bars from a component pallette onto initially blank windows. We now call such mechanisms "GUI builders" in the generic.

- Delphi became object-oriented from top to bottom. Precisely what this means is hard to explain before explaining the fundamentals of the Pascal

language. For now, just take it on faith. Think of object-orientation as a new, higher level of program structure that defines relationships between code and data in a hierarchical way.

- Delphi actually wrote parts of the code for new applications, especially applications crafted with the help of the GUI builder. Delphi basically generated a sort of generic program skeleton that programmers would "flesh out" to make it perform specific tasks. Object-orientation was essential to this code-generation mechanism.

Delphi was a roaring success for many reasons, primarily that it made the process of writing programs to run using Windows GUI elements easier to understand and hugely faster. The improvement in programmer productivity was such that Delphi became known as a *rapid application development* (RAD) environment.

Delphi is still being actively developed and is widely used by corporate developers, especially outside the US. It is, however, *very* expensive compared to the original $50 Turbo Pascal. Even the entry-level version of Delphi costs over $1000. This puts it beyond the means of newcomers and hobby programmers who don't have a corporation with an IT budget behind them.

Back when Delphi was released, a company called SpeedSoft offered a product called SpeedPascal, written to run under IBM's ill-fated OS/2 operating system. Speedsoft created a RAD environment for OS/2, using SpeedPascal as the "back end" compiler. The RAD environment was called Sibyl. Sibyl was very compatible with Delphi, and allowed programmers to port Delphi apps to OS/2 without rewriting all their code from scratch. OS/2's success began to wane in the shadow of Windows 95's success, and Sibyl's fate was tied to OS/2. In 1998, SpeedSoft released the source code for Sibyl and made it an open-source product before the company finally closed its doors.

With the Sibyl source code as a model, a group of Israeli programmers began a project in 1998 that they called Megido. The idea behind Megido was to create an open-source Windows-based RAD environment like Borland's Delphi. The project vanished after about a year for reasons that never became fully known. Later in 1999 several programmers resurrected the idea of Megido as the Lazarus Project. (Lazarus, some of you may recall from Sunday School, was the man whom Jesus brought back from the dead.)

The Sibyl source code was less useful than originally thought, largely because so much of it was in 32-bit x86 assembly and thus not easily portable to platforms based on other CPUs. So in recent years, Lazarus has been written entirely in FreePascal, thus making it at least potentially portable to any platform where FreePascal exists.

At this writing (April 2025) Lazarus is still very actively in development. Now that it has reached version 3.8 (with V4.0 in testing) it is definitely mature enough to develop commercial applications with.

Lazarus' major elements

Lazarus consists of a number of interdependent subsystems. Most of these will not come into play in this book, but you should be aware that they exist:

- A Pascal syntax-aware windowed source code editor.

- The FreePascal compiler.

- An interface to the Gdb debugger.

- The Lazarus Component Library, a code library roughly equivalent to the Delphi VCL.

- Interfaces to major GUI toolits and widget sets, including the Win32 and Win64 GDI, Gtk+ 1.2 and 2.6, and Max OS X Carbon. ("Widgets" are GUI controls like check boxes, pull-down menus, radio buttons, etc.) Others are in development, including Qt4.2 and Cocoa.

- The Code Explorer, which lets you quickly view the names of things in your programs without having to look at all the code. It creates a sort of "structure summary" of a program, allowing you to "take in" many details at once.

- The Object Inspector, which is a window providing a view of the sometimes subtle relationships among properties and methods within FreePascal objects.

- The Restriction Browser, which provides quick access to the restrictions implemented in the objects used in a program.

One thing to be aware of is that a lot of the way Lazarus is designed caters to the needs of object-oriented programming (OOP) and you'll need to learn the fundamentals of Pascal before confronting OOP at all. The Object Inspector and the Restriction Browser, in particular, make little sense until you "get" the object-oriented idea. The Lazarus Component Library is completely object-oriented and is not something that may be called from within very simple Pascal programs.

In fact, simple Pascal programs of the sort used in teaching can be created, compiled, and debugged under Lazarus using just three windows: The main window (which is a sort of expanded menu bar), the Source Editor window, and the Messages window, where status reports and errors from the compiler are displayed.

A project-oriented IDE

Through most of the history of Pascal programming, a "project" in Pascal was simply a Pascal program, or perhaps a program and a number of Pascal library files named within the program and linked with the program to create the executable program file. As Pascal programming has gotten more complex, and especially with the advent of object orientation and GUI apps for Windows and the Linux graphical shells, Pascal projects include more than simple Pascal code. You need icon files, form files, and many other things in the general category called *resources*. For this reason, when you want to write a program in FreePascal using Lazarus, you don't simply create a new Pascal source code file. You create a project.

A Lazarus project includes the main program's source code file, obviously. It also includes code libraries and other resources. Giving a project a single name (which does *not* have to be the name of any source code file) gives you a sort of umbrella beneath which you can tinker the various elements of an ambitious Pascal project without losing track of what parts belong to what project.

After creating a project, you can add new files to the project whenever you need to, and being able to treat the collection of files as a single named entity helps a lot in managing the complexity of a sophisticated project. I recommend that you give each project its own directory on disk to keep its files separate from those of other projects'.

Software conversations "beneath the surface"

Beginners sometimes wonder, after installing and looking at Lazarus and its many windows, where the window for the FreePascal compiler is. The truth is, the compiler does not need a window, and does not have one.

The Lazarus graphical environment can be thought of as a sophisticated "remote control" system for FreePascal. You type source code into the source code window, and select options in several of the other windows. When you have your project set up the way you want it (or at least the way you *think* you want it) you basically tell Lazarus to turn the FreePascal compiler loose on the project.

Lazarus runs the FreePascal compiler invisibly, passing to the compiler the necessary names and locations of pertinent files in your project, along with certain terse instructions as to how the project should be compiled and linked. The compiler does what it's told, and it passes back to Lazarus various messages indicating success or (failing complete success) error conditions. Lazarus digests these messages, and relays some of them up to you through the Messages window.

This process happens down where you can't watch it, and that's OK. Whatever messages coming back from FreePascal that you might find remotely useful will

appear in the Messages window. Note that not all messages appearing in the Messages window indicate that something is wrong. Many simply tell you what FreePascal is doing at that moment, and by displaying indicate that it succeeded.

5.2. CONFIGURING LAZARUS FOR PASCAL PROGRAMMING

Lazarus was designed to help you write GUI apps for Windows and other graphical user interface shells like GNOME and KDE under Linux. For learning the fundamentals of Pascal when you're first starting out, Lazarus may seem like overkill. It's possible to simplify Lazarus somewhat, to make it easier to grasp while you're learning it, and to keep unneeded portions of the system from distracting you while you're working.

Closing unneeded windows

Lazarus differs from most Windows and Linux GUI applications in that it consists of a varying number of disconnected windows distributed across the display. Some of its complexity lies in the sheer number of windows that open up when the Lazarus environment is in full roar. The first step in simplifying Lazarus is to close the windows that you won't need. To open, compile, and debug the non-OOP example programs presented in this book, you need the following windows to be open:

- The Main Window, which contains the main menu and the component palette. It's the wide but narrow window at the top of your screen. (This cannot be closed without closing Lazarus as a whole.)

- The Source Editor window. This is where you enter and edit your Pascal source code.

- The Messages window. This is where the FreePascal compiler will post status and error messages when you compile your programs.

Bringing one of Lazarus' windows into visibility is done from the menu bar in the main window. Pull down the View menu, and you'll see a list of items. Each item in the first group is a separate window. You make a window visible by selecting its item in the View menu.

Items in the View menu are *not* toggles. That is, you don't select an item once to show and then select it a second time to hide. To "put a window away" so that it's no longer visible, you click on the small "x" in the upper right corner of the window. Closing a window in that fashion changes nothing in the window itself; the window can be made visible again simply be selecting it from the View menu.

Configuring line numbering

By default, Lazarus does not display line numbers on every source window line. It can be configured to display numbers every *n* lines, where *n* can be any number from 1 to 100. When reading enormous source code files, having line numbers only once in ten might be useful, but when you're just learning, having a number on each line is better, especially since I will be referring to individual listing lines by number in this book.

Setting line numbering so that every line is numbered is easy. Do this:

1. From the Lazarus main menu, select **Tools | Options...** This brings up the IDE Options dialog.

2. From the treeview on the left side of the dialog, find the Editor option, and under Editor, select Display.

3. At the top of the pane labeled "Margin and Gutter" find the spinner control labeled "Every n-th line number."

4. Select the default value (which in Lazarus 1.2.0 is 5) and type 1. This means that every "1th" line will be numbered, which of course means all of them.

5. Click OK.

If you don't see any line numbers at all when you have the Source Editor open, make sure that the Show line numbers check box is checked. The check box is in the same pane as the "Every n-th line number" spinner control.

Changing the IDE language

If your first language isn't English, Lazarus will allow you to change the language in which some of the text in the IDE and most error messages are displayed. Please note that this feature is partial—and in my experience, a little buggy. The text in the main menus remains in English. In my tests (using Dutch and Spanish, two languages with which I'm a little familiar) the Spanish support was more complete. To change the IDE language, select **Tools | Options...** and click on the **General** item under the **Environment** header in the treeview. Under **Language** you'll see a drop-down list of all supported languages. Select the one you want and click OK.

I've had some trouble with language changes taking effect. The best way to make sure is to change the IDE language, and then close and reopen Lazarus. This seems to happen most often when you change a language away from the default (English) and then attempt to change it back again.

Setting up a project(s) directory

One thing I recommend before even starting Lazarus for the first time is to decide where your Pascal projects are going to "live" on your hard drive. I suggest creating a separate directory for them somewhere. To simplify backup, many people use the Windows My Documents directory tree for things that change frequently and need regular backup. The Linux Home directory serves the same purpose.

Here's a system that should work, at least while you're learning and your projects are relatively simple: Create a folder under My Documents (or under Home if you use Linux) called Lazarus Projects. Then, when you create a new Lazarus project, create a subdirectory under Lazarus Projects to hold a specific project. This can be done through the **Create New Folder** button on the **Save As...** dialog.

5.3. Creating a FreePascal Project in Lazarus

In this book I'll be talking about simple Pascal projects only. As I said earlier, these will be console applications, not Windows or Linux GUI apps. The FreePascal/Lazarus partnership can create several different sorts of projects, all of which may be written from within Lazarus and maintained by its project management machinery.

To create a new Pascal project in Lazarus, first close the default project that appears when you launch Lazarus. Next, select **Project | New Project...** in the main menu. The **Create a new project** dialog will appear. It displays the different kinds of projects that may be created in the left pane of the dialog.

Highlight **Simple Program** and click Ok. You may notice that there's another option in the left pane called **Console Application**. Simple FreePascal programs are also console applications. The **Console Application** option creates a more complex kind of program that is intended to be run in a system console but is based on an object-oriented framework. That's a far more complex business, and I can't cover it in this book. (I'm planning other books that will cover objects in detail.)

After you click Ok, Lazarus will display the Source Editor window, and place a "skeleton" program in it. For a brand-new project, it will look like Figure 5.1. Note that if you already had a project open in Lazarus, it will ask you if you want to save changes before displaying the new project. If you were just poking around Lazarus and weren't working on anything, you can discard changes.

Naming and saving your new project

Lazarus gives newly created projects a generic name: Project1, or Project2 (etc.) if a Project1 already exists. Even before you add a single line to the skeleton program of a new project, you should give it a "real" name and save it to disk.
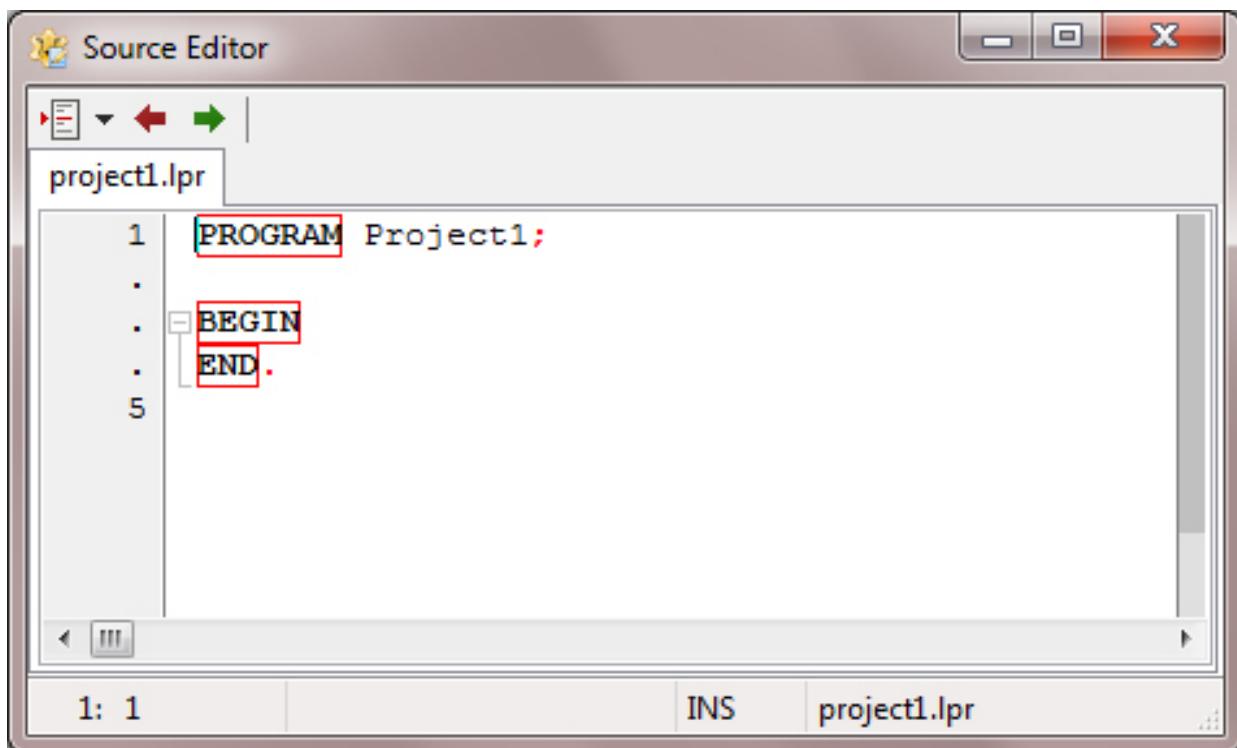
Figure 5.1. A simple Pascal program skeleton created by Lazarus.

This is done with the **Save As...** item under the **File** menu. If you intend to create a directory for the project at the same time that you save it to disk, it's a 2-step process:

1. Navigate to the parent directory (something like Lazarus Projects) and click the **Create New Folder** icon. Enter a descriptive name for the new folder that appears in the treeview and then click Enter.

2. Make sure that the name of the new folder is highlighted, and click **Open**. Lazarus will then "move" to the new folder.

3. Enter a name for your project and click **Save**. Done!

5.4. Using the Source Editor window

The main purpose of the Source Editor window is to allow you to enter and change your Pascal source code. Think of it as a simple word processor with some additional Pascal-specific machinery. It recognizes certain Pascal elements like constants and compiler directives, and puts them in distinctive colors. It knows (unless you confuse it!) which **BEGIN** reserved word goes with which **END**. It recognizes predefined identifiers and displays them in bold. It tells you which line of a file your cursor is on, and at which text column. It tells you whether it's in insert mode or overwrite mode.

Source Editor syntax display

Figure 5.1 shows the Source Editor opened. Note that the reserved words **program**, **begin**, and **end** are highlighted in boxes. This tells you the three reserved words are related: The **begin** and **end** define the block for the **program** reserved word. At any point in a program, you can click on any instance of **BEGIN** or **END**, and Lazarus will highlight what it thinks is the matching reserved word, which define a compound statement between them. (Remember that character case doesn't matter in Pascal, and that **BEGIN** and **begin** are the exact same thing!)

Note that this only works if your syntax is correct. It's possible to have extraneous or unmatched **BEGIN**s or **END**s in a source code file, which will confuse Lazarus and you as well. Compile the file to be sure there are no syntax errors before assuming the highlighting is correct.

Source Editor shortcuts

Navigating within the Source Editor window is done the same way it's done in most editing windows. Below is a summary of the Source Editor key commands for various tasks. Note that these are the shortcuts for Windows. Linux shortcuts will be similar, but may depend on the shell that you're using.

Arrow keys: Move one position for each keypress.

PgUp and PgDn: Move the cursor a number of lines that is roughly half the number of lines currently visible. In other words, if you have the Source Editor window open showing twenty lines, the PgUp and PgDn keys will move nine or ten lines up or down with each keypress.

Home: Move to the beginning of the current line.

End: Move to the end of the current line.

Ctrl-Home: Move to the beginning of the displayed file.

Ctrl-End: Move to the end of the displayed file.

Ctrl-Right arrow: Move one word right.

Ctrl-Left arrow: Move one word left.

Ctrl-T: Delete from the cursor position to the end of the current line.

Ctrl-Y: Delete the current line entirely.

Ctrl-Z: Undo the last command.

Ctrl-Shift-Z: Redo the last command.

If you have a mouse with a scroll wheel, the wheel will scroll the Source Editor window up or down with one line per wheel click. The cursor does *not* follow the scroll. This is useful to look up or down beyond the limits of the current screen without moving your cursor away from the point in the source code where you're currently working.

5.5. FreePascal Modes

Once you've gotten your Pascal legs and go beyond simple programs, you may discover that FreePascal doesn't like something you've written. One thing to investigate is whether you're using a feature not covered by the mode in which FreePascal is curently operating. FreePascal supports several different modes. All of them are basically compatibility modes, and dictate compatibility with other, often older versions of Pascal.

To use a specific mode, include the appropriate **MODE** compiler switch at the beginning of a program. The full text of each compiler switch is given after the name of the mode below:

FreePascal mode   {MODE fpc}

This is the default. If you don't tell FreePascal to operate in some other mode, this is the mode that it will use. Because it's the default mode, you don't have to include the **MODE fpc** directive to use FreePascal mode.

GNU Pascal mode   {MODE gpc}

This mode duplicates the dialect of Pascal understood by GNU Pascal.

Macintosh Pascal mode   {MODE macpas}

This mode provides compatibility with Macintosh Pascal dialects.

Turbo Pascal / Borland Pascal 7 mode   {MODE tp}

This mode provides compatibility with pre-Delphi Pascal dialects from Borland, especially Borland Pascal 7.
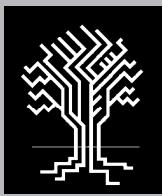
Lazarus mode   {MODE objfpc}

This is the mode that Lazarus uses when it compiles a GUI app created within the Lazarus environment. (I will not be covering GUI apps in this book.) It's somewhere between FreePascal mode and Delphi mode, and corrects some ambiguities in the dialect of Pascal used by Delphi.

Delphi mode   {MODE delphi}

This mode tries to be as compatible as possible with the latest release of Embarcadero's Delphi dialect of Pascal. The details of this mode change as Embarcadero evolves Delphi and the Delphi dialect of Pascal changes. If you're bringing projects over from Delphi (something I won't be covering at all in this book) this is the mode you should use.

PART III:

THE CORE OF THE PASCAL LANGUAGE

Having gotten a feeling for both programming and the "big picture" of Pascal itself, it's now time to after the details of both the Pascal language and the programming process. And therein lies a snag for me.

I'd like to be able to say, "Read this book from cover to cover and you'll learn Pascal." That's how I think books of this kind work best. The problem with teaching Pascal in a strictly linear fashion is that it's a continuous, linear narrative that weaves all of Pascal's ideas together in a single stream. There would be no "chapter" on simple data types, because you have to introduce simple data types right alongside reserved words, identifiers, operators, and so on. This reads well in the fashion of a novel, but it doesn't *re-read* very well. By that I mean that once you've read it, you may feel that your understanding of derived types is fuzzy, and you may want to go back and reread a discussion of derived types. But in a strictly linear exposition of Pascal, derived types are discussed in no one place, but here, there, and in several other places. Like all the other topics, they weave in now and again in the one big story.

Once I've laid the groundwork it'll be a lot easier. But in laying the groundwork I've had to make a number of seemingly arbitrary decisions: Do I explain first what reserved words and identifiers are, and then explain what data is? Or do I explain data types first, so that I can explain how reserved words and identifiers come together to make statements? Maybe it's a silly thing to worry about. But I'm trying hard to make this book accessible to total newcomers to the idea of programming, and putting across the foundations of *any* idea is absolutely critical.

So you'll have to cut me a little slack here. I've chosen to begin with Pascal's fundamental atoms like letters and symbols. In keeping with Pascal's idea of structure, I'll go from there to reserved words and identifiers, and from there to data types. After data types I'll cover operators, expressions, and statements. That's mostly a linear, bottom-up approach, and it will help things come back to mind when you return to these early chapters for review. Some might prefer that I cover operators before the apparently more complex topic of data types, but trust me: Data types without operators make more sense than operators without data types.

It will help a great deal if you've already read Chapter 3, which presents an overview of most of the fundamental ideas of Pascal. If you haven't, you ought to go back and read Chapter 3 now. The overview material will definitely help bridge the occasional chicken-and-egg dependencies you'll find in the next several chapters. And once you've got the core of the language under your belt, the problem goes away.

Really!

CHAPTER 6.
PASCAL ATOMS

Pascal, by design, is a structured language. Unlike certain "freeform" languages like APL and Perl, as well as older versions of BASIC and FORTRAN, it imposes a very clear structure on its programs. Pascal will not let you string statements together haphazardly, even if every statement, taken alone, is syntactically correct. There is a detailed master plan that every Pascal program must follow and the compiler is pretty strict about enforcing the rules. A program must be coded in certain parts. Some parts must go *here,* and others must go *there.* Everything must be in a certain order. Some things cannot work together. Other things *must* work together.

Aside from some concessions to compiler designers (the Pascal language specification makes their task easier in some respects) Pascal's structure exists solely to reinforce a certain way of thinking about programming. This way of thinking represents language creator Niklaus Wirth's emphasis on developing programs that are comprehensible without scores of pages of flowcharts and thousands of lines of explication. As I described in Part I, this way of thinking championed by Wirth and others is called "structured programming." Although structured programming can be accomplished in any computer language (even BASIC, though I have my doubts about APL) Pascal is one of a growing family of computer languages that absolutely *requires* it.

A structure must be made of *something*. A crystal is a structure of atoms in a particular orderly arrangement. A Pascal program is made of "atoms" that are formed from the ASCII character set, and "molecules" formed from ASCII characters. These program atoms and molecules fall into several categories. There are numbers and symbols. There is a fairly small group of *reserved words,* an even smaller number of *modifiers,* and then the unlimited multitude of ordinary identifiers created by you, the programmer, as names for your data items, procedures, and functions. In this chapter we'll discuss the different kinds of Pascal atoms and how they're used to create structured programs.

6.1. Symbols, Digits, and ASCII Characters

The most "atomic" of Pascal's atoms (and perhaps "subatomic") are the individual members of the ASCII character set. The American Standard Code for Information Interchange (ASCII) is a set of 128 values based on the English character set. Of the 128 ASCII characters, 94 are visible when displayed. 33 are not visible.

Of those 33 "invisible" characters, several are called *whitespace* characters, which do not have associated glyphs (symbols) but which give structure to visible text on the page or screen by dividing text into lines and aligning text within lines vertically into structures. Whitespace characters include the space character, the tab character, the carriage return character, and the line feed character. Whitespace characters do divide your Pascal programs into words and lines, but they are mostly there for your benefit: Without them, your programs would be one long line of symbols and words spanning many pages, and would be virtually impossible to read. The FreePascal compiler strips all whitespace out except for the space characters that separate the visible elements of the program from one another.

When you write program code, it looks like this on your display and printouts:

```
FOR I := 1 TO 17 DO
  BEGIN
    Shr(J);
    Inc(J)
  END;
```

However, after it removes excess whitespace, the compiler "sees" a continuous and unformattted stream like this:

```
FOR I := 1 TO 17 DO BEGIN Shr(J); Inc(J) END;
```

The point here is that how you arrange the lines of code in your program is not important to the FreePascal compiler. You can indent lines by two spaces per level (as is my custom) or six or eight spaces. You can place two lines in between procedures and functions, or three, or whatever you prefer. You can, if you like, place short compound statements (that is, code falling between **BEGIN** and **END** reserved words) on a single line. Pascal reserved words and identifiers must be separated from one another by spaces. Beyond that, well, it's up to you.

Some of the invisible 33 ASCII characters are "control characters" that were originally designed to control the operation of mechanical teletype printers and later on, CRT terminals. These are almost all obsolete nowadays and rarely used. Interestingly, one such control character, the BEL character, was originally used to ring a small mechanical bell in teletype machines, and later on caused CRT terminals to beep.

Numeric Digits

Individually and together, the digits 0-9 are used by Pascal as you would expect: to express literal numeric values:

```
8
71
29784
```

Pascal does not separate large values into groups of thousands with commas, as we often do when writing ordinary text; for example, 29,784. Placing a comma within a number will trigger a compiler error.

In addition to expressing literal numeric values, the digits 0-9 can also be part of the identifiers you create as variable and procedure names; for example, **Area51**. More on this a little later.

Alphabetic characters

The alphabetic characters A-Z and a-z are used individually and in combination to create the names of compiler commands, reserved words, constants, variables, procedures, and functions. Note well that unlike some other programming languages (the C familiar, particularly) character case is not significant. For example, the FreePascal compiler considers **A** and **a** to be the same character except when present in quoted string literals. As I'll explain later, within a string literal, case is significant; **'Jeff'** and **'JEFF'** are considered different by the compiler.

Symbols

The ASCII character set includes a number of visible symbols that are neither numeric digits nor letters of the English alphabet. These include punctuation marks like ! and ?, arithmetic symbols like + and -, straight and curly brackets **[ ] { }**, slashes, and a few others. Most such symbols have very specific meanings to the FreePascal compiler. Some symbols can mean more than one thing; for example, the period character, which expresses decimal parts of numeric literals, and also references to fields within records. Sometimes two symbols are combined into a single symbol. For example, the two symbols **<>** together mean "not equal to" and the symbols **(*** begin a comment.

For the most part, symbols are not allowed within identifiers. That is, you cannot create a variable named **Ham&Cheese**. The exception is the underscore character; **Ham_and_Cheese** is perfectly legal. More on this later as well.

6.2: RESERVED WORDS: PASCAL'S FRAMING MEMBERS

Reserved words are words that have special meanings within the Pascal language. They cannot be used by the programmer except to stand for those particular meanings. The compiler will immediately error-flag any use of a reserved word that is not rigidly in line with that word's meaning. Examples would include **BEGIN**, **END**, **PROCEDURE**, **FUNCTION**, **ARRAY**, and that old devil **GOTO**.

Which words the FreePascal compiler considers reserved words depends on what mode you have the compiler operating in. As I explained a little earlier in Chapter 5, FreePascal can work in several modes. In each mode, FreePascal "works like" a different Pascal compiler. Those modes are controlled by the **$MODE** switch. The two modes you're likely to use as a beginning programmer are Turbo Pascal mode and FreePascal mode. If you begin using the Lazarus GUI Builder to create graphical apps (which I will not be covering in this book) you will then be operating in Delphi mode. FreePascal mode is the default. To work in Turbo Pascal mode you must set the **$MODE TP** switch. Table 6.1 lists all the reserved words defined in Turbo Pascal mode.

Table 6.1. Reserved Words enforced in Turbo Pascal 7.0 mode

ABSOLUTE	ELSE	NIL	SET
AND	END	NOT	SHL
ARRAY	FILE	OBJECT	SHR
ASM	FOR	OF	STRING
BEGIN	FUNCTION	ON	THEN
BREAK	GOTO	OPERATOR	TO
CASE	IF	OR	TYPE
CONST	IMPLEMENTATION	PACKED	UNIT
CONSTRUCTOR	IN	PROCEDURE	UNTIL
CONTINUE	INHERITED	PROGRAM	USES
DESTRUCTOR	INLINE	RECORD	VAR
DIV	INTERFACE	REINTRODUCE	WHILE
DO	LABEL	REPEAT	WITH
DOWNTO	MOD	SELF	XOR

Table 6.2 lists the additional reserved words recognized by FreePascal when operating in Delphi mode, using the **$MODE DELPHI** switch. Note well that these are *in addition* to the reserved words in Table 6.1. When operating in Delphi mode, FreePascal recognizes all of the reserved words in Table 6.1 as well as all the reserved words in Table 6.2.

Table 6.2. Additional reserved words enforced in Delphi mode.

AS	FINALIZATION	LIBRARY	RAISE
CLASS	FINALLY	ON	THREADVAR
EXCEPT	INITIALIZATION	OUT	TRY
EXPORTS	IS	PROPERTY	

When FreePascal is operating in its default FreePascal mode (or has been set to FreePascal mode using the $MODE FPC switch) it enforces a few more reserved words in addition to all of those enforced in Turbo Pascal 7 mode and Delphi mode. These are listed in Table 6.3. Again, keep in mind that when operating in FreePascal mode, the compiler enforces *all* reserved words listed in Tables 6.1 through 6.3.

Table 6.3. Additional reserved words enforced in FreePascal mode.

DISPOSE	FALSE	TRUE
EXIT	NEW	

In addition to the reserved words shown in Tables 6.1-6.3, there is another list of predefined words that you should be aware of. These are called *modifiers,* and they work in conjunction with certain reserved words to modify what those reserved words mean to the compiler, hence their name. For the most part, you will have no call to use them while you're just learning FreePascal. However, it is important that you *not* use them for anything else in your own programs.

Table 6.4. Modifiers understood by FreePascal

ABSOLUTE	EXTERNAL	NOSTACKFRAME	READ
ABSTRACT	FAR	OLDFPCCALL	REGISTER
ALIAS	FAR16	OVERRIDE	SAFECALL
ASSEMBLER	FORWARD	PASCAL	SOFTFLOAT
CDECL	INDEX	PRIVATE	STDCALL
CPPDECL	LOCAL	PROTECTED	VIRTUAL
DEFAULT	NAME	PUBLIC	WRITE
EXPORT	NEAR	PUBLISHED	

The compiler has its own somewhat arcane uses for them, but it will also allow you to use them if you choose to, as the names of variables, constants, procedures, or functions. That, however, is a bad idea. I recommend treating the list of modifiers in Table 6.4 as though they were additional reserved words, especially when you're just starting out. To avoid confusion, especially as you advance in your skills, use

them *only* where and how they were designed to be used by FreePascal's authors. All of them are considered advanced to *very* advanced topics, and I will be discussing few if any of them in this first book. For now, treat them as a list of words to be avoided as you define named elements in your own programs.

Put your reserved words in uppercase!

In the Ancient Days of Pascal, standard practice was always to place reserved words in upper case, and I'll do so throughout this book. Pascal is not case-sensitive, so you can place reserved words in lower or mixed case if you wish. **BEGIN** is treated the same way by the compiler as **begin** or **Begin** or (for that matter) **BeGIn**. In recent years, a strange aversion to upper case characters has appeared among programmers, and in most other books about Pascal and Delphi you will see reserved words and nearly everything else in lower case.

That's a mistake. Uppercase helps reserved words stand out like beacons when you read your own code listings or those of others. Because reserved words are the framing members of your Pascal programs and govern the very shape that your programs take, I think it pays to be able to see them clearly, at a glance. *This is important*, and although I'm considered "old-school" (and a bit of a crank) for insisting on it, I make no apologies and will no longer discuss the issue. Putting everything in your program in lower case makes it hard to read, and from my perspective, readability is the #1 feature of correct Pascal code. Basically, even if your Pascal code is bug-free, if it's hard to read, it's worthless.

6.3. Identifiers

Your computer creates programs for your use. Programs are collections of things (data) and steps to be taken in storing, changing, and displaying those things (statements). The computer knows such things by their addresses in its memory. The readable, English-language names of data, of programs, of functions and procedures, are for your benefit. We call such names, taken as a group, *identifiers*. They exist in the source code only. *Identifiers do not exist in the final executable code file.*

Any name that *you* invent and apply to an entity in a program is an identifier. The names of variables, of data types, of named constants, of procedures and functions, and of the program itself are identifiers. An identifier you create can mean whatever you want it to mean (within Pascal's own rules and limits) as long as it is unique within its scope. There are also identifiers that the compiler predefines, and there are identifiers defined in code libraries that FreePascal allows you to use.

The notion of scope is a subtle and important concept that I will explain at length later on, but broadly put, it indicates what portion of your program the compiler can "see" at any given point in your code. The compiler will complain if it can "see" more than one item with the same identifier. That is, if you have a variable named **Counter** you cannot have a procedure named **Counter** within the same scope. Nor can you have another variable (or anything else) named **Counter** within that scope. Understanding scope requires that you understand more about program structure than I've explained so far; hold on until we get there.

FreePascal identifiers are sequences of characters of any length up to 255 characters that obey these few rules:

- Legal characters include letters, digits, and underscores. Spaces and symbols like &,!,*, or % (or any symbol not a letter or digit) are not allowed.

- Digits (0-9) may *not* be used as the first character in an identifier. All identifiers must begin with a letter from A-Z (or lowercase a-z) or an underscore.

- Identifiers may not be identical to any reserved word.

- Case differences are ignored. The letter **A** is the same as **a** to the compiler.

- Underscores are legal and significant. Note that this differs from most older, non-Borland Pascal compilers, in which underscores are legal but ignored. You may use underscores to "spread out" a long identifier and make it more readable: **Sort_On_ZIP_Code** rather than **SORTONZIPCODE**. (A better method that has become the custom is to use "camel case" to accomplish the same thing: **SortOnZIPCode**.)

- *All* characters are significant in a FreePascal identifier, up to 255 characters. Some ancient Pascal compilers allowed identifiers of arbitrary length but ignored any character after the eighth character.

The following are all invalid identifiers that will generate errors:

`Fox&Hound`	Contains an invalid character, "&", that may not be in an identifier
`FOO BAR`	Contains a space
`7Eleven`	Begins with a number
`RECORD`	RECORD is a reserved word and can't be used as an identifier

Here's a point that Pascal beginners sometimes misunderstand: There is no penalty at all for using a long identifier over a short identifier. In other words, the identifier **I** and the identifier **StatusOfPrimaryKeyUpdateOperation** take up precisely the same amount of space in the compiled program that FreePascal

generates. And that amount of space is…none! *Identifiers do not exist in your executable binary program.* They are used by FreePascal to build a "symbol table" of memory addresses that allow it to generate the executable code file, but identifiers are left behind when the work is done.

With that in mind, create identifiers that help you read your source code. That's what they're for. An identifier can be too long, of course, but that's rarely the problem.

A *quick look ahead by looking back: Variable definition*

I'm going to do something here that in programming is called a "forward reference": I'm going to briefly explain how identifiers become variables. There's a chicken-and-egg problem in describing Pascal fundamentals, in that you need variables to demonstrate operators, and you need operators to define variables. I'll explain in a great more detail how data items are defined in the next chapter, but for now, think back to what you learned in Section 3.3. In fact, re-reading that section now wouldn't be a bad idea.

To recap: In Pascal, *variables must be defined before they're used.* Defining a variable in Pascal is done using the **VAR** reserved word, up toward the top of your program. The **VAR** reserved word associates an identifier with a data type. For example:

```
VAR
  I : Integer;
```

This is a complete Pascal statement. It creates a variable by associating a valid identifier (here, the letter "I") with a standard Pascal data type, **Integer**. Your program will now treat variable **I** as an integer, and allow **I** to be used in certain kinds of numeric operations expressed in Pascal statements. If you attempt to use **I** in ways outside the powers of the **Integer** type, FreePascal will call foul, and you'll get a type conflict error at compile time. Variable definition in Pascal is an important and subtle business, I'll have much more to say on it in Chapter 7.

6.4. Operator Basics

In Pascal, an *operator* is a symbol or short group of symbols that specifies either a relationship between two data items, or else some kind of operation to be done on one or more values, hence the name. Some Pascal operators are single characters, like "=", but because there are only so many characters in the ASCII character set, most operators are short groups of two or three characters, like ">=" and "**NOT**". An operator operates on one or more variables or constants. These variables and constants are called the operator's *operands*.

A good place to start is with the *assignment operator*. A variable is a container in memory laid out by the compiler. It has a particular size and shape defined by its type. You load an item of data into a variable, being certain that it conforms to the size and shape of the variable. This data is the variable's *value*.

You're probably already familiar with the assignment operator, which is perhaps the most fundamental operator in Pascal: "**:=**" The assignment operator is (usually) how values are placed into variables. Consider this simple assignment statement:

```
I := 17;
```

A value on the right side of the assignment operator is assigned to the variable on the left side of the assignment operator. In an assignment statement, there is always a variable on the left side of the assignment operator. On the right side may be a constant, a variable, or an expression.

Expressions

An *expression* in Pascal is a combination of data items and operators that eventually "cook down" to a single value. Data items are constants and variables. The best way to understand expressions is to think back to your grade-school arithmetic, and how you used arithmetic operations to combine two or more values into a single result. This is an expression, both in basic arithmetic and in Pascal:

```
17 + 3
```

The addition operator + performs an add operation on its two operands, 17 and 3. The value of the expression is 20.

An expression like "**17 + 3**", while valid in Pascal, would not be used in a real program, where the literal value "20" would suffice. It's a lot more useful to create expressions that involve variables. For example:

```
Pi * Radius * Radius        { Pi is a predefined constant in FreePascal }
```

This expression's value is recognizable as the area of the circle defined by whatever value is contained in the variable **Radius**. Note here that a Pascal expression is *not* the same thing as a Pascal statement, and expressions do not stand alone. To pass the compiler's picky standards, an expression must always be part of a statement.

Standard Pascal includes a good many different operators for building expressions, and FreePascal enhances Standard Pascal with a few additional operators. They fall into a number of related groups depending on what sort of result they return: Relational, arithmetic, set, string, and logical (also called "bitwise") operators.

6.5. Relational operators

The relational operators are used to build Boolean expressions, that is, expressions that evaluate down to a Boolean value of **True** or **False**. Boolean expressions are the most widely used of all expressions in Pascal. All of the looping and branching statements in Pascal depend on Boolean expressions. More on this in Chapter 7.

A relational operator causes the compiler to compare its two operands for some sort of relationship. The Boolean value that results is calculated according to a set of well-defined rules as to how data items of various sorts relate to one another.

Table 6.5 summarizes the relational operators implemented in FreePascal. All return Boolean results:

Table 6.5. Relational Operators

Operator	Symbol	Operand types	Precedence
Equality	=	scalar, set, string, pointer, record, object	5
Inequality	<>	scalar, set, string, pointer, record, object	5
Less than	<	scalar, string	5
Greater than	>	scalar, string	5
Less than or equal	<=	scalar, string	5
Greater than or equal	>=	scalar, string	5
Set membership	IN	set, set members	5
Set inclusion, left in right	<=	set	5
Set inclusion, right in left	>=	set	5
Negation	NOT	Boolean	2
Conjunction	AND	Boolean	3
Disjunction	OR	Boolean	4
Exclusive OR	XOR	Boolean	4

The column labeled "Precedence" has to do with order of evaluation, which I'll return to later on in this book.

The set operators fall into two separate worlds; they're set operators, obviously, but they also express certain relationships between sets and set members that return Boolean values. (I'll come back to sets later on.) The convention is that any operator that returns a Boolean value is relational, since Boolean values express the "truth or falsehood" of some stated relation. The three relational operators that involve sets, set membership and the two set inclusion operators, will be discussed along with the

operators that return set values, in Section 8.3. Note that the set inclusion operators share symbols with the greater than or equal to/less than or equal to operators, but the *sense* of these two types of operations is radically different.

Some of the types mentioned briefly in this section are types I haven't yet explained in detail. Most of these will be covered in the next chapter except for pointers, which will have to wait until considerably later on. The mentions are here not so much for you to read on your *first* linear pass through this book, but on those occasions when you return to this section for a brushup.

Equality

If two values compared for equality are the same, the expression will evaluate as **True**. In general, for two values to be considered equal by FreePascal's runtime code, they must be identical on a bit-by-bit basis. This is true for comparisons between like types. Most comparisons must be done between values of the same type.

The exceptions are comparisons done between numeric values expressed as different types. FreePascal allows comparisons rather freely among integer types and real number types, but this sort of type-crossing must be done with great care. In particular, do not compare calculated reals (real number results of a real number arithmetic operation) for equality, either to other reals or to numeric values of other types. Rounding effects may cause real numbers to appear unequal to compiled code even though the mathematical sense of the calculation would seem to make them equal.

Integer types **Byte**, **ShortInt**, **Word**, **Integer**, and **LongInt** may be freely compared among themselves.

Two sets are considered equal if they both contain exactly the same members. (The two sets must, of course, be of the same base type to be compared at all.) Two pointers are considered equal if they both point to the same dynamic variable. Two pointers are also considered equal if they both contain the predefined value **NIL**.

Two records are considered equal if they are of the same type (you cannot compare records of different types) and each field in one record is bit-by-bit identical to its corresponding field in the other record. Remember that you *cannot* compare records, even of the same type, using the greater than/less than operators >, <, >=, or <=.

Two strings are considered equal if they both have the same logical length (see Section 8.6) and contain the same characters. This makes them bit-by-bit identical out as far as the logical length, which is the touchstone for all like-type equality comparisons under Turbo Pascal. Remember that this makes leading and trailing blanks significant:

```
'    Eriador'  <> 'Eriador'     { True }
'Eriador     ' <> 'Eriador'     { True }
```

Inequality

The rules for testing for inequality are exactly the same as the rules for equality. The only difference is that the Boolean state of the result is reversed:

```
17 = 17                { True  }
17 <> 17               { False }
42 = 17                { False }
42 <> 16               { True  }
```

In general, you can use the inequality operator anywhere you can use the equality operator.

Pointers are considered unequal when they point to different dynamic variables, or when one contains the value **NIL** and the other does not. The bit-by-bit rule is again applied: Even one bit's difference found during a like-type comparison means the two compared operands are unequal. The warning applied to rounding errors produced in calculated reals applies to inequality comparisons as well.

Greater Than/Less Than

The four operators greater than, less than, greater than or equal to, and less than or equal to, add a new dimension to the notion of comparison. They assume that their operands always exist in some well-defined order by which the comparison can be made.

This immediately disqualifies pointers, sets, and records. Saying one pointer is greater than another simply makes no sense, given the way pointers are defined. You could argue that since pointers are physical addresses, one pointer will always be greater or less than another non-equal pointer. This may be true, but in the spirit of the Pascal language, details about how pointers are implemented are hidden from the programmer at the purely Pascal level. Messing around with pointers beneath the surface can get you into various kinds of trouble, and shouldn't be attempted until you're aware of the issues—and even then, perhaps not at all.

The same applies to sets and records. Ordering them makes no logical sense, so operators involving an implied order cannot be used with them.

With scalar types (see Section 7.7) a definite order is part of the type definition. For integer and cardinal types (that is, types that express whole numbers) the order is obvious from our experience with arithmetic; integers and cardinals model whole numbers between specific bounds.

The **Char** and **Byte** types are both limited to 256 possible values, and both have an order implied by the sequence of binary numbers from 0 to 255. The **Char** type is ordered by the ASCII character set, which makes the following expressions evaluate to **True**:

```
'A' < 'B'
'a' > 'A'
'Z' < 'a'
'@' < '['
```

The higher 128 values assignable to **Char** variables have no truly standard character glyphs outside of the PC world, but they still exist in fixed order and are numbered from 128 to 255.

Enumerated types are limited to no more than 255 different values, and usually have fewer than ten or twelve. Their fixed order is the order the values were given in the definition of the enumerated type:

```
TYPE
  Colors = (Red,Orange,Yellow,Green,Blue,Indigo,Violet);
```

This order makes the following expressions evaluate to **True**:

```
Red < Green
Blue > Yellow
Orange > Red
Indigo < Violet
```

The ordering of string values involves two components: The length of the string and the ASCII values of the characters present in the string. Essentially, FreePascal begins by comparing the first characters in the two strings being compared. If those two characters are different, the test stops there, and the ordering of the two strings is based upon the relation of those two first characters. If the two characters are the same, the *second* characters in each string are compared. If they turn out to be the same, then the third characters in both strings are compared.

This process continues until the code finds two characters that differ, or until one string runs out of characters. In that case, the longer of the two is considered to be greater than the shorter. All of the following expressions evaluate to **True**:

```
'AAAAA' > 'AAA'
'B' > 'AAAAAAAAAAAA'
'AAAAB' > 'AAAAAAAAAA'
```

NOT, AND, OR, and XOR

There are four operators that operate on Boolean operands: **NOT**, **AND**, **OR** and **XOR**. These operators are sometimes set apart as a separate group called Boolean operators. In some ways, they have more in common with the arithmetic operators than with the relational operators. They do not test a relationship which already exists between two operands. Rather, they combine their operands according to the rules of Boolean algebra to produce a new value that becomes the value of the expression.

The simplest of the four is **NOT**, which takes only one Boolean operand. The operand must be placed after the **NOT** reserved word. **NOT** negates the Boolean value of its operand:

```
NOT False       { Expression is True  }
NOT True        { Expression is False }
```

Some slightly less simplistic examples:

```
NOT (6 > I)     { True for I > 6   }
NOT (J = K)     { True for J <> K  }
```

The parentheses indicate that the expression within the parentheses is evaluated first, and only then is the resultant value acted upon by **NOT**. This expression is an instance where "order of evaluation" becomes important. I'll discuss this in detail in Section 6.7.

AND (also known as "conjunction") requires two operands, and follows this rule: *If both operands are* **True**, *the expression returns* **True**; *else the expression returns* **False**. If either operand or both operands have the value **False**, the value of the expression as a whole will be **False**. Some examples:

```
True AND True                 { Expression is True  }
True AND False                { Expression is False }
False AND True                { Expression is False }
False AND False               { Expression is False }

(7 > 4) AND (5 <> 3)          { Expression is True  }
(16 = (4 * 4)) AND (2 <> 2)   { Expression is False }
```

All of these example expressions use constants, and thus are not realistic uses of **AND** within a program. We present them this way so the logic of the statement is obvious without having to remember what value is currently in a variable present in an expression. We'll be presenting some real-life coding examples of the use of **NOT**, **AND**, and **OR** in connection with the discussion of order of evaluation in Section 6.7.

OR (also known as "disjunction") requires two operands, and follows this rule: *If either (or both) operands is **True**, the expression returns **True**; only if both operands are **False** will the expression return **False**.*

Some examples, again using constants:

```
True OR True              { Expression is True  }
True OR False             { Expression is True  }
False OR True             { Expression is True  }
False OR False            { Expression is False }


(7 > 4) OR (5 = 3)        { Expression is True  }
(2 < 1) OR (6 <> 6)       { Expression is False }
```

Finally, there is **XOR**, which also requires two operands, and follows this rule: *If both operands are the same Boolean value, **XOR** returns **False**; only if the operands have unlike Boolean values will **XOR** return **True**.* Some examples:

```
True XOR True             { Expression is False }
True XOR False            { Expression is True  }
False XOR True            { Expression is True  }
False XOR False           { Expression is False }
```

Greater Than or Equal To/Less Than or Equal To

These two operators are each combinations of two operators. These combinations are so convenient and so frequently used that they were welded together to form two single operators with unique symbols: **>=** (read, "greater than or equal to",) and **<=** (read, "less than or equal to.")

When you wish to say:

```
X >= Y
```

you are in fact saying

```
(X > Y) OR (X = Y)
```

and when you wish to say

```
X <= Y
```

you are in fact saying

```
(X < Y) OR (X = Y)
```

The rules for applying >= and <= are exactly the same as those for < and >. They may take only scalars or strings as operands.

6.6: Arithmetic operators

Manipulating numbers is done with arithmetic operators, which along with numeric variables, form arithmetic expressions. About the only common arithmetic operator not found in Standard Pascal is the exponentiation operator, that is, the raising of one number to a given power. FreePascal has one, however, and we'll build a function that raises one number to the power of another in Section 11.4.) Table 6.6 summarizes the arithmetic operators implemented in FreePascal.

Table 6.6. Arithmetic Operators

Operator	Symbol	Operand Types	Result Type	Precedence
Addition	+	Integer, real, cardinal	same	3
Sign Inversion	-	Integer, real	same	1
Subtraction	-	Integer, real, cardinal	same	3
Multiplication	*	Integer, real cardinal	same	2
Integer Division	DIV	Integer, cardinal	same	2
Real Division	/	Integer, real, cardinal	real	2
Modulus	MOD	Integer, cardinal	same	2
Exponentiation	**	Integer, cardinal	same	1

Note that for the purposes of the table, "Integer" types include **Integer**, **LongInt**, **ShortInt**, and **SmallInt**. "Cardinal" types include **Word**, **Byte, LongWord, Cardinal**. "Real" types include **Real**, **Single**, **Double, Extended,** and **Currency.** The old **Comp** type inherited from Borland Pascal is also considered a real type, but it is not implemented on all platforms and I recommend that you not use it. It was necessary at the time (back in the 1980s!) but there are better and more standard real number types to use today.

There are a lot of numeric types, and there are "gotchas" connected with some of them. I'll explain numeric types in a great deal more detail in the next chapter.

The Table 6.6 "operands" column lists those data types that a given operator may take as operands. The compiler is fairly free about allowing you to mix types of numeric variables within an expression. In other words, you may multiply bytes by integers, add reals to bytes, multiply integers by bytes, and so on. For example, these are all legal expressions in FreePascal:

```
VAR
  I,J,K  : Integer;
  R,S,T  : Real;
  A,B,C  : Byte;
  U,V    : Single;
  W,X    : Double;
  Q      : ShortInt;
  L      : LongInt;

I * B             { Integer multiplied by byte }
R + J             { Integer added to real      }
L * Q             { LongInt multiplied by ShortInt }
C + (R * I)       { Etc. }
J * (A / S)
```

The "result type" column in the table indicates the data type that the value of an expression incorporating that operator may take on. Pascal is ordinarily very picky about the assignment of different types to one another in assignment statements. This "strict type checking" is relaxed to some extent in simple arithmetic expressions. Numeric types may, in fact, be mixed fairly freely within expressions as long as a few rules are obeyed:

1. Any expression including a floating point value may only be assigned to a floating-point variable.

2. Expressions containing floating point division (/) may only be assigned to a floating point variable *even if the operands are integer types.*

Failure to follow these rules will generate a type mismatch error. However, outside of the two restrictions above, a numeric expression may be assigned to any numeric variable, assuming the variable has sufficient range to contain the value of the expression. (More on range in the next chapter.) For example, if an expression evaluates to 14,000, you should not assign the expression to a variable of type **Byte**, which can only contain values from 0 to 255. Program behavior in cases like that is unpredictable. If range checking is on, such an assignment will generate a range error.

Addition, subtraction, and multiplication are handled the same way ordinary arithmetic is handled with pencil or calculator. Division is a little trickier.

Division

There are three distinct division operators in Pascal. One supports floating point division, and the other two support division for integer and cardinal types. Floating point division (/) may take operands of any numeric type, but it always produces a

floating point value, complete with decimal part. Attempting to assign a floating point division expression to an integer type will generate error a type mismatch error, *even if all numeric variables involved are integers.*

```
VAR
  I,J,K : Integer;

I := J / K;        { Won't compile!!! }
```

Why? Dividing two integers can generate a decimal part, and type **Integer** cannot express a decimal part.

Division for numbers that cannot hold decimal parts is handled much the same way division is first taught to grade schoolers: When one number is divided by another, two numbers result. One is a whole number quotient; the other a whole number remainder.

In Pascal, integer division is actually two separate operations that do not depend upon one another. One operator, **DIV**, produces the quotient of its operands:

```
J := 17;
K := 3;
I := J DIV K;     { I is assigned the value 5  }
```

No remainder is generated at all by **DIV**, and the operation should not be considered incomplete. If you wish to compute the remainder, the modulus operator **MOD** is used:

```
I := J MOD K;     { I is assigned the value 2  }
```

Assuming the same values given above for **J** and **K**, the remainder of dividing **J** by **K** is computed as 2. The quotient is not calculated at all (or calculated internally and thrown away); only the remainder is returned.

Sign inversion

Sign inversion is a "unary" operator; that is, it takes a single operand. What it does is reverse the sign of its operand. It will make a positive quantity negative, or a negative quantity positive. It does not affect the absolute value (distance from zero) of the operand. Sign inversion can only be used with signed numeric types, which include the real number types plus **ShortInt**, **SmallInt**, **Integer**, **LongInt**, and **Int64**.

Note that sign inversion *cannot* be used with cardinal types like **Cardinal**, **Byte**, **Word, or LongWord**. Cardinal types are "unsigned"; that is, they are never considered negative, and so changing the sign of the value is impossible. I'll have a

whole lot more to say about FreePascal's numeric types in Chapter 7.

6.7. Bitwise Operators

There is a whole class of FreePascal operators that won't be of much use to you as a beginner, but for completeness' sake they're worth describing here. These are the *bitwise operators*. The bitwise operators were not originally present in the Standard Pascal language. Borland's Turbo Pascal introduced them in the early 1980s.

In some uses data is "bit-mapped"; that is, certain bits have certain meanings apart from all other bits and must be examined, set, and interpreted individually; that is, one single bit at a time. The way to do this is is through the bitwise logical operators. Associated with the bitwise logical operators are the shift operators, **SHL** and **SHR**. We will speak of these shortly.

We have previously spoken of the **AND**, **OR**, **XOR**, and **NOT** operators, which work on Boolean operands and return Boolean values. The bitwise logical operators are another flavor of **NOT**, **AND**, **OR**, and **XOR**. They work with operands of all integer types (**Integer**, **Word**, **LongInt**, **ShortInt**, and **Byte**) and they apply a logical operation upon their operands, done one bit at a time. Table 6.7 summarizes the bitwise logical operators and the shift operators:

Table 6.7. The bitwise logical and shift operators

Operator	Symbol	Operand Types	Result Type	Precedence
Bitwise NOT	NOT	All integer types	same	2
Bitwise AND	AND	All integer types	same	3
Bitwise OR	OR	All integer types	same	4
Bitwise XOR	XOR	All integer types	same	4
Shift Right	SHR	All integer types	same	3
Shift Left	SHL	All integer types	same	3

The best way to approach all bitwise operators is to work in true binary notation, where all numbers are expressed in base two, and the only digits are 1 and 0. The bitwise operators work on one binary digit at a time. The result of the various operations on 0 and 1 values is best summarized by four "truth tables":

```
    NOT              AND              OR              XOR
 NOT 1 = 0      0 AND 0 = 0     0 OR 0 = 0      0 XOR 0 = 0
 NOT 0 = 1      0 AND 1 = 0     0 OR 1 = 1      0 XOR 1 = 1
```

```
1 AND 0 = 0     1 OR 0 = 1     1 XOR 0 = 1
1 AND 1 = 1     1 OR 1 = 1     1 XOR 1 = 0
```

When you apply bitwise operators to two 8-bit or two 16-bit data items, it is the same as applying the operator between each corresponding bit of the two items. For example, the following expression evaluates to **True**:

```
$80 = ($83 AND $90)     { All in hexadecimal }
```

Why? Think of the operation $83 & $90 this way:

```
Hex        Binary
$83 =  1 0 0 0 0 0 1 1
           AND
$90 =  1 0 0 1 0 0 0 0
           =
$80 =  1 0 0 0 0 0 0 0
```

Read *down* from the top of each column in the binary number, and compare the little equation to the truth table for bitwise **AND**. If you apply bitwise **AND** to each column, you will find the bit pattern for the number $80 to be the total result.

   Now, what good is this? Suppose you only wanted to examine four out of the eight bits in a variable of type **Byte**. The bits are numbered 0-7 from the right. The bits you need are bits 2 through 5. The way to do it is to use bitwise **AND** and what we call a "mask:"

```
VAR
  GoodBits, AllBits : Byte;

GoodBits := AllBits AND $3C;
```

To see how this works, let's again "spread it out" into a set of binary numbers:

```
AllBits     = X X X X X X X X
                    AND
$3E (mask) =  0 0 1 1 1 1 0 0
                    =
GoodBits    = 0 0 X X X X 0 0
```

Here, "X" means "*either* 1 or 0." Again, follow the eight little operations down from the top of each column to the bottom. The zero bits present in four of the eight columns of the mask, $3C, force those columns to evaluate to zero in **GoodBits**, regardless of the state of the corresponding bits in **AllBits**. Go back to the truth table if this is not clear: *If either of the two bits in a bitwise AND expression is zero, the result will be zero.*

This way, we can assume that bits 0,1,7 and 8 in **GoodBits** will always be zero and we can ignore them while we test the others.

Shift operators

We've looked at bit patterns as stored in integer types, and how we can alter those patterns by logically combining bit patterns with bitmasks. Another way to alter bit patterns in integer types is with the shift operators, **SHR** and **SHL**. **SHR** stands for SHift Right; **SHL** for SHift Left.

Both operators are best understood by looking at a bit pattern before and after the operator acts upon it. Start with the value $CB (203 decimal) and shift it two bits to the right as the **SHR** operator would do:

```
 1 1 0 0 1 0 1 1 -->
  \ \ \ \ \ \
  \ \ \ \ \ \
--> 0 0 1 1 0 0 1 0
```

The result byte is $32 (50 decimal). The two 1-bits on the right end of the original $CB value are shifted off the end of the byte (into the "bit bucket," some say) and are lost. To take their place, two 0-bits are fed into the byte on the left end.

SHL works identically, but in the other direction. Let's shift $CB to the left with **SHL** and see what we get:

```
<-- 1 1 0 0 1 0 1 1
     / / / / / /
     / / / / / /
   0 0 1 0 1 1 0 0 <--
```

Again, we lose two bits off the end of the original value, but this time they drop off the left end, and two 0-bits are added in on the right end. What was $CB is now $2C (44 decimal.)

Syntactically, **SHL** and **SHR** are like the arithmetic operators. They act with the number of bits to be shifted to form an expression, the resulting value of which you assign to another variable:

```
Result := Operand <SHL/SHR> <number of bits to shift>;
```

Some examples:

```
VAR
  B,C : Byte;
  I,J : Integer;
```

```
I := 17;
J := I SHL 3;     { J now contains 136 } B := $FF;
C := B SHR 4;     { C now contains $0F }
```

It would be a good exercise to work out these two examples shifts shown above on paper, expressing each value as a binary pattern of bits and then shifting them.

An interesting note on the shift operators is that they are extremely fast ways to multiply and divide a number by a power of two. Shifting a number one bit to the left multiplies it by two. Shifting it two bits to the left multiplies it by four, and so on. In the example above, we shifted 17 by three bits, which multiplies it by 8. Sure enough, 17 X 8 = 136.

It works the other way as well. Shifting a number one bit to the right divides it by two; shifting two bits to the right divides by four, and so on. The only thing to watch is that there is no remainder on divide and nothing to notify you if you overflow on a multiply. It is a somewhat limited form of arithmetic, but in time-critical applications you'll find it is *much* faster than the more generalized multiply and divide operators.

6.7. ORDER OF EVALUATION

Mixing several operators within a single expression can lead to problems for the compiler. Eventually the expression must be evaluated down to a single value, but in what order are the various operators to be applied? Consider the ambiguity in this expression:

```
7 + 6 * 9
```

How will the compiler interpret this? Which operator is applied first? As you might expect, there are rules that dictate how expressions containing more than one operator are to be evaluated. These rules define what we call *order of evaluation*.

To determine the order of evaluation of an expression, the compiler must consider three factors: Precedence of operators, left to right evaluation, and parentheses.

Precedence

All operators in Pascal have a property called *precedence*. Precedence is a sort of evaluation prioritizing system. If two operators have different precedences, the one with higher precedence is evaluated first. There are five degrees of precedence: 1 is the highest and 5 the lowest.

When we summarized the various operators in tables, the rightmost column contained each operator's precedence. The sign inversion operator has a precedence of one. *No other*

operator has a precedence of one. Sign inversion operations are always performed before any other operations, assuming parentheses are not present. (We'll get to that shortly.) Logical and bitwise **NOT** operators have a precedence of two. For example:

```
VAR
  OK,FileOpen : Boolean;

IF NOT OK AND FileOpen THEN CloseFileWithError;
```

How is the expression, "**NOT OK AND FileOpen**" evaluated? **NOT OK** is evaluated first, because **NOT** has a precedence of 2, whereas **AND** has a precedence of 3. (See Table 6.1 for the precedence of these and other relational operators.) The Boolean result is then **AND**ed with the Boolean value in **FileOpen**, to yield the final Boolean result for the expression. If that value is **True**, the procedure **CloseFileWithError** is executed.

Left to Right Evaluation

The previous example was clear-cut since **NOT** has a higher precedence than **AND**. But as you can see from the tables, many operators have the same precedence value; addition, subtraction, and set intersection are only a few of the operators with a precedence of 4, and most relational operators have a precedence of 5.

When the compiler is evaluating an expression and it confronts a choice between two operators of equal precedence, it evaluates them in order from left to right. For example:

```
VAR
  I,J,K : Integer;

J := I * 17 DIV K;
```

The ***** and **DIV** operators both have a precedence of 3. To evaluate the expression **I * 17 DIV K**, the compiler must first evaluate **I * 17** to an integer value, and then integer divide that value by **K**. The multiplication operator ***** is to the left of **DIV**, and so it is evaluated before **DIV**.

Note that left to right evaluation happens *only* when it is not clear from precedence (or parentheses, see below) which of two operators must be evaluated first.

Setting Order of Evaluation with Parentheses

There are situations in which the previous two rules break down. How would the compiler establish order of evaluation for this expression:

```
I > J AND K <= L
```

The idea here is to test the Boolean values of two relational expressions. The precedence of **AND** is greater than the precedence of any relational operator like **>** and **<=**. So the compiler would attempt to evaluate the subexpression **J** AND **K** first.

Actually, this particular expression does not even compile; FreePascal will flag an error as soon as it finishes compiling **J AND K** and sees another operator ahead of it.

The only way out of this one is to use parentheses. Just as in the rules of algebra, in the rules of Pascal, parentheses override *all* other order of evaluation rules. To make the offending expression pass muster, you must rework it this way:

```
(I > J) AND (K <= L)
```

Now the compiler first evaluates **(I > J)** to a Boolean value, then **(K <= L)** to another Boolean value, then submits those two Boolean values as operands to **AND**. **AND** happily generates a final Boolean value for the entire expression.

This is one case (and a fairly common one) in which parentheses are *required* in order to compile the expression without errors. However, there are many occasions when parentheses will make an expression more readable, even though, strictly speaking, the parentheses are not required:

```
(Pi * Radius) + 7
```

Here, not only does * have a higher precedence than **+**, ***** is to the left of **+** as well. So in any case, the compiler would evaluate **Pi * Radius** before adding 7 to the result. The parentheses make it immediately obvious what operation is to be done first, without having to think back to precedence tables and consider left to right evaluation.

I'm a bit of a fanatic about program readability. Which of these (identical) expressions is easier to dope out:

```
R + 2 * Pi - 6
R + (2 * Pi) - 6
```

You have to think a little about the first. You don't have to think about the second at all. I powerfully recommend using parentheses in all but the most completely simpleminded expressions to indicate to all persons (including those not especially familiar with Pascal) the order of evaluation of the operations making up the expression. Parentheses cost you *nothing* in code size or code speed. Nothing at all.

To add to your program readability, that's dirt cheap.

6.8. STATEMENTS

Put as simply as possible, a Pascal program is a series of statements. Each statement is an instruction to do something: to define data, to alter data, to execute a function or procedure, to change the direction of the program's flow of control.

Perhaps the simplest of all statements is an invocation of a procedure or function. FreePascal includes a library containing several screen-control procedures for use in text program in console windows. These procedures enable your programs to move the text cursor around, clear the screen, and so on. Procedures in Pascal are generally used by naming them, and naming one constitutes a statement:

```
ClrScr;
```

This statement tells the computer to do something; in this case, to clear the screen. The computer executes the statement; the screen is cleared, and control passes to the next statement, whatever that may be.

We have been using assignment statements, type definition statements, and variable declaration statements all along. By now you should understand them reasonably well. Type definition statements must exist in the type definition part of a program, procedure, or function. They associate a type name with a description of a programmer-defined type. Enumerated types (see Section 8.2) are an excellent example:

```
TYPE
  Spectrum    = (Red,Orange,Yellow,Green,Blue,Indigo,Violet);
  LongColors  = Red..Yellow;
  ColorSet    = SET OF Spectrum;
```

Each of these three definitions is a statement. The semicolons *separate* the statements rather than terminate them. This is a critical distinction that frequently escapes beginning Pascal programmers. I'll take the vexing matter of semicolons up again in Section 9.8.

Variable declaration statements are found only in the variable declaration part of a program, function, or procedure. They associate a variable name with the data type the variable is to have. The colon symbol (**:**) is used rather than the equal sign:

```
VAR
  I,J,K : Integer;
  Ch    : Char;
  R     : Real;
```

Assignment statements are used in the body of the program, function, or procedure. They copy data from the identifier on the right side of the assignment

operator to the variable on the left. Only a *variable* can be to the left of the assignment operator. Constants, literals, and expressions must be on the right side. Other variables, of course, can also be to the right of the assignment operator:

```
I := 17;                { '17' is a numeric Literal }
R := Pi;                { Pi was defined earlier as a constant }
J := (17*K) + Ord(Ch);  { J is assigned from an expression }
K := J;                 { The value of one variable assigned to another }
```

This is only a quick description of the simplest statement types. We'll return in detail to the topic of statements in Chapter 9, when we confront Pascal's "flow control" machinery: Loops and conditional statements.

Recap: Pascal atoms, Pascal molecules, and Pascal organisms

As with everything in Pascal, there's structure going on here: Digits, letters, and symbols might be thought of as Pascal's "subatomic particles." Combined according to Pascal's rules, you get reserved words, modifiers, identifiers, and operators, which might be considered Pascal's atoms.

Combining reserved words, operators, and identifiers gives you statements. If you like the metaphor, statements might be considered Pascal's molecules. For example, **IF** and **THEN** and are reserved words. The equal sign = is a symbol, and in nearly all cases in Pascal also acts as an operator. **GotoXY** is the identifier of a library function. **Counter**, **Limit**, and **NextProg** are programmer-defined (that is, defined by you) identifiers.

This little snippet of Pascal code is a statement:

```
IF Counter = Limit THEN GotoXY(5,7);
```

Statements intuitively come across as "molecules of action," much as English-language sentences are. The statement above is almost sentence-like: "If Counter equals Limit, then GotoXY to position 5,7." Combining multiple statements in a fashion that respects Pascal's structure gives you programs, which can be thought of (stretching the metaphor perhaps a little beyond usefulness) as Pascal "machines" or even "organisms." The idea you need to take away from the metaphor is that in Pascal, big things are made from smaller things, which in turn may be made from even smaller things, according to rules that the compiler enforces. The atoms, for the most part, are givens, but you make up the molecules, and from the molecules create even larger and more complex things. Obviously, we'll be talking about this in more detail throughout the book. But for our next step, we need to talk about data, and how Pascal defines data and structures it into data types.

CHAPTER 7.
DATA AND DATA TYPES

Data is the raw material of computing. We sometimes seem obsessed by the coding details of the programs that manipulates our data, but the data itself is what we buy or sell and ultimately live by. The machinery of programming lies empty without it. Good programs coalesce around a solid, detailed vision of what data should be, both coming into a process and leaving it. So let's look at the idea of data, and the way that the Pascal language defines and manipulates it.

7.1. THE NOTION OF TYPES AND TYPE CONFLICTS

Data are chunks of information that your program manipulates. There are different *types* of data, depending on what the data is intended to represent. How your program handles your variables depends completely on what type you decide they are. Every variable used in a Pascal program must be declared to be of some type, with a notation like this:

```
CreditHours : Integer;
```

The variable's identifier comes first, followed by a colon and then the name of the type that you choose to define that variable by. The definition shown above allows you to manipulate integer values in a variable called **CreditHours**, subject to Pascal's explicit limitations on what can be done with **Integer** data types.

I didn't show it explicitly in the above definition, but variable definitions must reside in the *variable definition section* of a Pascal program. The reserved word **VAR** begins the variable definition section, and it runs until another program section (say, a procedure or a function, or the main program) begins. The variable definition section of a program (and there may be more than one in FreePascal) is where you give names to variables and assign them types. We'll speak more of the several sections of a Pascal program when we dig deep into program structure later on. In the meantime, you can refer back to a complete Pascal program with a variable definition section on Page 72.

The nature of types

At the lowest level, all data of any type in a Pascal program (or in any program, really) is stored as one or more binary numbers somewhere in your computer's memory. The data type of a data item to some extent dictates the way those binary numbers are arranged in memory, and to a greater extent dictates how you as a human being will use that data. In Pascal, the type of a data item is actually a set of rules governing the storage and use of that data item. Data takes up space in memory. The type of a data item dictates how much space is needed and how the data is represented in that space. A signed integer, for example, occupies two bytes of memory. The most significant bit of an integer carries the sign (that is, whether the value is positive or negative) of the number that the integer represents.

The data type also governs how a data item may be used. The simplest example: What is the letter "A" plus 17? That's a meaningless question, since a letter is not the same sort of a thing as a number. They're used in different ways and "mean" in different ways when we think about them. In Pascal, a letter is of type **Char** (short for "character," obviously) and 17 is type **Integer**, from the technical term for a number (either positive or negative) with no decimal part. Pascal's rules of typing prevent you from adding two incompatible types like **Char** and **Integer**. The compiler will not compile any such attempt, and will give you a type conflict error for trying.

Char and **Integer** are of different sizes in memory, so there is a sort of natural incompatibility between them. However, the size of a type has very little to do with the rules that govern it. Type **Char** and type **Byte** are both stored as single 8-bit bytes in memory, but you can add or multiply two variables of type **Byte**. Attempting to use variables of type **Char** in an arithmetic expression will generate an error at compile time.

The Pascal language incorporates *strong typing*, which means that there are strict limitations on how individual types may be used, and especially on how variables of one type may be assigned to variables of another type. In most cases, a variable of one type may not be assigned to a variable of a different type. (The major exception is with numeric types, which have a broad compatibility with one another. We'll discuss that issue in more detail a little later.) Transferring information between variables across type boundaries is usually done with "transfer functions" that depend upon well-defined relationships between types. Transfer functions will be described in Chapter 11.

Simple types (which include all the types described in this chapter) are "unstructured;" that is, they are data atoms that cannot be broken down into simpler data types.

Pascal's fundamental data types

All Pascal compilers implementing the standard Pascal language share certain fundamental data types. These types express some of the most basic concepts of computing: text, numbers, and logical truth and falsehood. From these fundamental types you can build data structures to model practically any data "idea" you can think of, as I'll discuss in more detail in Chapter 8. Here are those fundamental types:

- *Integers* are numbers (including negative numbers) that cannot have decimal points, like 1, -17, and 4529. The category as I mean it here includes types representing both signed and unsigned numbers. FreePascal has a lot more numeric types than Standard Pascal. I'll describe them individually below.

- *Characters* (indicated by predefined identifier **Char**) consist of the ASCII character codes from 0 to 127. These include all the common letters, numbers, and symbols used by all modern computers. FreePascal extends the definition of type **Char** to include those first 128 ASCII characters plus an additional 128 characters containing non-English language characters, mathematical symbols, and the "line-draw" characters used to make text-mode boxes on the screen back when text displays were all we had. Don't be confused by the fact that the ASCII character codes are numbered. **Char** is not a numeric type; just because Box #3 has a number doesn't make it a number rather than a box!

- *Boolean* types have only two possible values, **True** and **False**. They represent the notion of truth and falsehood in logic, and are sometimes called "flags". Pascal uses them in conditional statements like **IF/THEN/ELSE** and **REPEAT/UNTIL** to determine whether to take some action or not. Boolean types will become very important in Chapter X, where I'll explain how you can use Boolean values and Pascal's control statements to produce structured programs.

- *Real* type variables are used to express "real numbers"; that is, numbers that may include a decimal part. 1.16, -3240, 6.338 and -74.0457 are all real numbers.

FreePascal's whole-number data types

In addition to these elementary data types understood by all Pascal compilers, FreePascal adds several of its own. Most of these are numeric types. They're numerous enough (and similar enough) to require a little discussion:

- **Byte** is a numeric type that can contain values from 0 to 255. **Byte** is an example of an *unsigned* or *cardinal* numeric type, meaning that it cannot

take on a negative value. That is, it can have neither a positive nor a negative sign in front of it, as can most of the standard numeric types. **Byte** values are always assumed to be positive. **Byte** gets its name from the fact that it occupies one single byte in memory. (The **Word** type occupies two bytes, and some of FreePascal's more esoteric numeric types occupy as many as ten.)

- **ShortInt** is a numeric type that resembles **Byte** in that it occupies one byte of memory. Unlike **Byte**, it can represent signed values, in the range -128..127. Type **ShortInt** is, in fact, a "signed byte."

- **SmallInt** is "small integer" and serves as a signed numeric type that can represent values in the range -32768..32767. It occupies two bytes in memory.

- **Word** is like **Byte** in that it is unsigned and cannot hold negative values. Like **SmallInt** it occupies two bytes of memory. **Word** may express values in the range 0..65535.

- **LongInt** is what its name implies: A "long integer." It occupies four bytes in memory rather than two, as the **Word** and **SmallInt** types do. Its range is hugely greater, however, and can express values from -2147483648 to 2147483647. Sidenote: You *cannot* use commas to break up the expression of large numbers in Pascal, as we often do in ordinary day-to-day correspondence. So while spitting out a number like 2147483647 seems awkward without the commas, do get used to it: it's simply the way we do things in programming.

- **LongWord** is related to **LongInt**, in that it occupies four bytes in memory. It is unsigned, and may express positive values in the range 0..4294967295.

- **Int64** is a signed type and can hold both negative and positive values. It occupies eight bytes in memory, or 64 bits, hence its name. Take a deep breath: Its range is -9223372036854775808 on the low side (negative nine quintillion!) to 9223372036854775807 on the high side.

- **QWord** is an unsigned type that occupies eight bytes in memory. It's the unsigned partner of **Int64**. Its range is 0..18446744073709551615. At some point naming huge numbers becomes numbing and meaningless, but just for fun, that's zero to eighteen quintillion.

This list does not include the real number types in FreePascal. In programming, *real numbers* are those containing a decimal part, like 3.14159. There are several real number types in FreePascal , and I'll address them in detail later, in Section 7.7.

How big is a FreePascal integer?

People with some experience in Pascal may notice that the fundamental type **Integer** is not given on the previous list of whole number types. Nor is the type **Cardinal**, which Delphi programmers may recall from their work with Borland's environment. The reason is this: Depending on the underlying processor (and to some extent, the operating system) the Pascal **Integer** type may be defined as either **SmallInt (16 bits)** or **LongInt** (32 bits).

In other words, when you define a variable of type **Integer** in a program, the range of the variable, and the number of bytes it occupies in memory, may not always be the same. This sounds crazy, but it's necessary. FreePascal was designed to allow compilation of programs to run on all kinds of different CPUs, including those designed for embedded systems work, which may not have the internal machinery to process 32-bit integer values quickly.

This presumably includes Intel legacy processors like the 8086 and 8088, which are 16-bits in size at the register level. I would test them except that I haven't had a machine that old around the house in many years!

All that said, at this writing (2014) FreePascal treats definitions of type **Integer** as 32-bit **LongInt** on both 32-bit and 64-bit Intel x86 CPUs. If you're generating code for a different CPU (perhaps for embedded systems work) you'll probably be aware of the CPU's register width. If not, well, you'd better look it up.

The **Cardinal** type is included in FreePascal for compatibility with Delphi code, and in FreePascal is always treated as type **LongWord**.

With great range come problems

As so often happens in technical work, there are gotchas. The two types **Int64** and **QWord** are *not* ordinal types. They're perfectly at home with bogglingly large numeric values, but certain Pascal built-in functions will not work with them. The two functions **Pred** and **Succ** are the ones you're most likely to encounter as a newcomer. (I discuss these in Section 11.6; look ahead if you're interested.) In short, whole number values have a successor value (i.e., the "next" value up) or a predecessor value (the next value "down") and these can be tested by the **Pred** and **Succ** functions...*unless* the value was defined as **Int64** or **QWord**. Then, the compiler will hand you an error.

In 64-bit CPUs, memory pointers are 64 bits in size. In 32-bit CPUs, pointers are 32 bits in size. If you're doing pointer math, your pointers and your integers must be the same size. This is a perilous business but sometimes must be done. If so, use the numeric types **PtrInt** (signed) and **PtrUInt** (unsigned) on the numeric side.

7.2. SIMPLE CONSTANTS

Constants are data values that are "baked into" your source code and do not change during the execution of a program. There are two kinds of simple constants in Standard Pascal: literal and named. FreePascal adds a third kind of constant that is not really a constant: *typed constants,* that is, constants that have a specified type and can be data structures like arrays and records. I'll discuss typed constants in Section 8.7, after I've had a chance to explain data structures.

Literal versus named constants

A *literal constant* is a value of some sort that is stated as a value where it is used in your code. For example:

```
SphereVolume := (4/3)*Pi*(Radius*Radius*Radius);
```

In this line of code, "4" and "3" are literal constants, representing the numeric values of 4 and 3.

There is another constant in that statement: The identifier **Pi,** which is predefined in FreePascal as the real number value 3.14159265358979. You can define your own constants as you need them. The constant declaration part begins with the reserved word **CONST** and runs until some other part of the program begins. (You do not need to end the **CONST** section with **END.**) The constant declaration part is typically very early in a program, and is often the very first part of a program after the program name itself:

Here, **Pi** denotes a *named constant.* We could as well have used the literal constant 3.14159 in the statement, but "**Pi**" is shorter and makes the expression less cluttered and more readable. And more precise: FreePascal's **Pi** constant goes out 14 places! Especially where numbers are concerned, named constants almost always make a program more readable.

Another use for constants is in the setting of program parameters that need changing only very rarely. They still *might* be changed someday, and if you use a named constant, changing the constant *anywhere in the program* is only a matter of changing the constant's declaration *once* in the constant declaration part of the program and then recompiling.

The alternative is to hunt through hundreds or thousands of lines of source code to find every instance of a literal constant to change it. You will almost certainly miss at least one, and the resultant bug explosion may cost you dearly in time and torn hair.

In short, don't use literal constants anywhere you will *ever* anticipate needing changes. In mathematical formulae literal constants are usually OK; but keep in mind that using a named constant in place of a literal constant allows you to control the precision of the constant everywhere in the program, by the use of a single constant definition. You may want to use pi to eight decimal places initially, but later on, to improve program performance, you may decide that five decimal places is plenty. If you define the mathematical constant pi in a named constant at the front of your program, you can change the precision instantly just by changing the definition—and not have to worry about forgetting one or two places where you had hard-coded a specific value of pi into an expression.

Constants and their types

In Standard Pascal, constants may be simple types, strings, and sets only. (I'll be covering sets in detail in Section 8.3.) That group includes real numbers, integers, characters, strings, sets, and Booleans. Individual enumerated types may also be considered constants, although they are not declared the same way other constants are. (Like sets, enumerated types are derived types that I'll cover in Section 8.2.) Structured types like records, pointers and arrays may *not* be constants in Standard Pascal. FreePascal also supports data structure constants, (see Section 8.7) but I must explain data structures first before going into those.

Here are some example named constants of various types:

```
CONST
  Threshold    = -71.47;       { Negative FP real    }
  PenIOAddress = $06;          { Hexadecimal value   }
  Using8087    = True;         { Boolean             }
  DriveUnit    = 'A';          { Character           }
  Revision     = 'V2.61B';     { String              }
  Answer       = 42;           { Integer             }
  NotAnswer    = -42;          { Negative integer    }
  YesSet       = ['Y','y'];    { Set; See Chapter 8  }
  NullString   = '';           { Null (empty) string }
  BigNumber    = 6117834       { Long integer or real }
```

I haven't said much about strings yet (and will cover them in detail in Chapter 8) but string literals are easy to understand: You enclose some sequence of ASCII characters between single-quote marks:

```
'Call me Ishmael. In fact, call me anything but late for dinner.'
```

Literal string constants like this may be assigned to string variables later on in the program.

Constants versus variables

How is a constant different from a variable? The obvious difference is that the value of a constant is set at compile time. You cannot assign a value to a simple constant in an assignment statement. Given the list of example constants shown above, you could not legally code:

```
Answer := 47;
```

because **Answer** has been defined as a constant that already has a value of 42.

There is an important difference between constants and variables. In FreePascal, simple constants are written into the code by the compiler as what we call *immediate data*. This is difficult to explain if you don't have some grasp of how computers work down at the silicon level, but think of it this way: The value represented by a simple constant is written into your program every place you use it. If you were to define the **Answer** constant as shown above and reference it twelve times in your program, the compiler would store **Answer**'s equivalent binary value (here, 42) twelve times in your program code.

Variables, by contrast, are kept separate from the code portion of a program, and each variable is stored in only one place. Your program has a *data segment* in memory where it stores its variables, and a named variable is present in the data segment only *once*. To access the value stored in a variable, your program must use a memory reference into the data segment to the place where the variable's value is stored. This isn't something you need to know a lot about, and (especially while you're a beginner) where a variable is stored in memory and how the variable is represented in memory aren't terribly important. The important thing to remember is this:

- A constant is defined at compile time and cannot be changed while your program is running.

- A variable is not given a value until your program gives it one at runtime, and that value may be changed by your program at any point while the program runs.

The type of a constant depends, to some extent, on its context. Consider:

```
PROGRAM AllTheAnswers;

CONST
  Answer = 42;

VAR
  Tiny   : Byte;       { One byte                 }
  Little : ShortInt    { A short integer (1 byte) }
  Small  : Integer;    { An integer (2 bytes)     }
```

```
  Big    : LongInt      { A long integer (4 bytes)  }
  Huge   : Real;        { A real number (8 bytes)   }

BEGIN
  Tiny   := Answer;
  Little := Answer;
  Small  := Answer;
  Big    := Answer;
  Huge   := Answer;
END.
```

In the code snippet given above, **Answer**'s value is defined as 42. But it is perfectly legal to assign the value of **Answer** to type **Byte**, type **Integer**, type **ShortInt**, type **LongInt**, or type **Real**. The code the compiler generates to do the assignment in each case is a little different, and that code is smart enough to translate the numeric value "42" into a binary number that will be properly expressed as type **Byte**, **Integer**, **ShortInt**, **LongInt**, or **Real**, or any other numeric type supported by the compiler. But the end result is that all five variables of five different types will each express a numeric value of 42 in its own fashion.

Notes on literal constants

A dollar sign ($) in front of a numeric literal means that the compiler will interpret the literal as a hexadecimal number—that is, a number written in base 16. The numeric literal may *not* have a decimal point if it is to be considered hexadecimal. Hexadecimal digits, in case you're not familiar with the concept, run from 0 through 15, but the values 10 through 15 are expressed as the letters A (10) B (11) C (12) D (13) E (14) and F (15.) So a hexadecimal number may include both letters and digits:

```
$47
$5A
$B62F
$ECF6AA
```

I cover the hexadecimal system and hex numbers in great detail in my print book, *Assembly Language Step By Step, Third Edition* (John Wiley & Sons, 2009).

Inside string literals, lower case and upper case characters are distinct. If you wish to include a single quote mark as a character inside a string literal, you must use two single quotes together:

```
Writeln('>>You haven''t gotten it right yet...');
```

This line of code will display the following line:

```
>>You haven't gotten it right yet...
```

7.3. Constant expressions

Turbo Pascal 5.0 introduced the concept of *constant expressions* to the Pascal language, and FreePascal inherited that idea. Prior to Turbo Pascal 5, a named constant could be defined in only one way: By stating its name and equating it to a single literal value with the "=" symbol. For example:

```
FudgeFactor = 17;        { Simple named constant definition}
```

A constant expression allows you to give a value to a named constant in terms of an expression that is evaluated at compile time. Note that only simple constants may be given values from constant expressions. *Typed constants may not be assigned values from constant expressions.* Typed constants are a slightly odd concept, and I'll discuss them in Section 8.7.

An expression, briefly, is a combination of identifiers, values, and operators that "cooks down" to a single value. Expressions resemble portions of equations from physics, into which you plug necessary values and finally evaluate to a single value:

Kinetic energy = Mass * Velocity^2

Here, "Mass * Velocity^2" is an expression, albeit not one written in Pascal. Once you know the mass and velocity in the current context, you can plug those values into the equation and "do the math" to come up with a value for the kinetic energy.

It's much the same way with constant expressions. In a constant expression, a constant is given a value in terms of a combination of operators, values, and constants that were defined earlier in the program:

```
DropletDiameterInMM = 3;
DropletRadiusInMM = DropletDiameterInMM / 2;
DropletAreaInSqMM = 4 * Pi * DropletRadiusInMM * DropletRadiusInMM;
```

Here, **DropletDiameterInMM** is a simple named constant. Both of the constants that follow it are defined by constant expressions. When the compiler encounters a constant expression during its compilation of the program, it "does the math" and assigns the resulting value to the constant itself. A constant expression may make use of constants defined by earlier constant expressions. In the example above, **DropletRadiusInMM** is calculated from **DropletDiameterInMM**. On the next line, **DropletAreaInSqMM** is calculated from **DropletRadiusInMM**. The identifier **Pi** here is a named constant that is predefined by FreePascal and is always available to your programs.

As with simple constants, a value is calculated for a constant expression at compile time, and the constant may not be given a different value at runtime.

Limitations of constant expressions

If you're familiar with expressions as they occur in normal Pascal statements, you may be wondering if any legal expression may be assigned to a constant. The answer is emphatically *no*; the permissible expressions used in constant expressions are *much* more limited. Legal elements of a constant expression are these:

- *Literal constants.* These include numeric literals like 42 and 17.00576, the Boolean literals **True** and **False**, and quoted string and character literals like **'Z'** and **'Snark'**.

- *Previously defined constants.* In other words, a constant expression may make use of named constants defined earlier in the program, like **DropletDiameterInMM** in the example above.

- *Pascal operators.* These are the basic arithmetic operators like addition and subtraction, the logical operators like **AND** and **OR**, and the bitwise operators like **AND**, **OR**, **SHR**, **SHL**, and so on. For a complete discussion of FreePascal's operators, see Chapter 6.

- *Certain built-in functions.* A very few of FreePascal's built-in functions may take part in constant expressions. These include **Ord**, **Chr**, **Odd**, **Hi**, **Lo**, **Length**, **Abs**, **Pred**, **Succ**, and **Swap**. All other functions, including those you write yourself and those built into the compiler (including **Sqr** and **Cos**) are illegal. For those of you who can appreciate the differences at this point in your understanding of Pascal, FreePascal generates code for "functions" like **Abs** in-line (in the fashion of an assembly language macro) while other functions like **Sqrt** and **Cos** are true Pascal functions that must be called like any other functions from the FreePascal runtime library. **Abs** and its kin are more properly *macros* than functions, and can be evaluated in-line by the compiler during compilation.

Although it might be obvious to veteran Pascal programmers, it's worth stating clearly that *variables cannot be part of constant expressions*. FreePascal allows constants to be defined after variables are declared (as Standard Pascal does not) but not after variables are assigned values. This means that a variable would introduce an undefined quantity into a constant expression, which can cause a *lot* of trouble at runtime.

Some examples of constant expressions

The best way to show what's possible with constant expressions is to put a few in front of you. The following examples are all legal in Pascal, if not necessarily useful in every case:

```
CONST
  Platter   = 1;
  FirstSide = Odd(Platter);          { Boolean }
  FlipSide  = NOT FirstSide;         { Boolean}

  Yesses    = ['Y','y'];             { Character set; See Chapter 8 }
  Noes      = ['N','n'];             { Character set; See Chapter 8 }
  Answers   = Yesses + Noes;         { Union of 2 sets }

  USHoller  = 'ATTENTION!!  ';       { String }
  USGasMsg  = 'Fuel level is low!';  { String }
  USGasWarn = USHoller + USGasMsg;   { String concatenation }
  USWarnSiz = Length(USGasWarn);     { String length }

  LongSide  = 17;
  ShortSide =  6;
  TankDepth =  8;
  Volume    = LongSide * ShortSide * TankDepth; { Constant expression }
```

Why use constant expressions?

There are two excellent reasons to use constant expressions: Reconfiguration and documentation. Both relate to the use of what I call "magic numbers" in program development.

Many programs use values that may be defined once in a program and are never modified. These include mathematical constants, and I/O port numbers and bit numbers for low-level control of embedded systems hardware. These "magic numbers" aren't expected to change, but to be safe, they should always be defined as constants rather written out separately as dozens or hundreds of numeric literals shotgunned throughout what may be a very large program.

An excellent example of reconfiguration through constant expressions involves directly programming the communications port on the original PC. This used to be fairly common in the DOS era, when direct control of PC hardware was easy and virtually unrestricted. Windows and Linux now forbid such application-level tinkering, but direct access to hardware like serial ports is still done in embedded systems work. FreePascal is sometimes used to develop embedded systems code, although I won't be covering that use in this book.

Whether or not you ever get a chance to work on serial port driver code, as an example it's still instructive: Two serial ports, COM1: and COM2:, are supported by the PC hardware. They are accessed through different sets of I/O ports, and the differences in the port addresses follows an unchanging relationship. By defining a single constant specifying either COM1: or COM2:, the control port addresses may be recalculated in constant expressions based on that single port number definition. This is what the following code does:

```
CONST
  ComPort = 1;        { 1 = COM1:  2 = COM2: }
  ComBase = $2F8;     { Build on this "magic number" }

  { Base I/O port is $3F8 for COM1: and $2F8 for COM2: }
  PortBase = ComBase OR (ComPort SHL 8);

  { Transmit Holding Register is write-only at the base port: }
  THR = PortBase;

  { Receive Buffer Register is read-only at the base port: }
  RBR = PortBase;

  IER = PortBase + 1;  { Interrupt Enable register }
  IIR = PortBase + 2;  { Interrupt identification register }
  LCR = PortBase + 3;  { Line control register }
  MCR = PortBase + 4;  { Modem control register }
  LSR = PortBase + 5;  { Line status register }
```

Don't panic if most of this doesn't make immediate sense to you! What matters here is knowing how each of the identifiers in the example above is given a value.

Every one of the constants defined in the example code fragment has a different value depending on whether the COM1: or COM2: serial ports is to be used. By changing the value of the constant **ComPort** from 1 to 2, *all* the other constants change accordingly to the values that apply to serial port COM2:. The program does not need to be peppered with magic numbers like $2FC and $3FA. Also, your program does not need to spend time initializing all these port numbers as variables, because the compiler does all the calculation at compile time, and the resulting values are inserted as immediate data into the code generated from your source file.

The other use for constant expressions helps your programs document themselves. You may need some sort of mathematical "fudge factor" in a complicated program. You can define it as a simple named real-number constant:

```
FudgeFactor = 8.8059;
```

No one, looking at the literal numeric value, would have any idea of its derivation. If the value is in fact the result of an established formula, it can help readability to make the formula part of a constant expression:

```
ZincOxideDensity = 5.606;
FudgeFactor = ZincOxideDensity * (Pi / 2);
```

This will help others (or maybe even you) keep in mind that you had to fudge things by multiplying the density of zinc oxide by pi over 2. (Note: I've deliberately made this "fudge factor" ridiculous. Many are.) The idea should never be far from your mind that

Pascal programs are meant to be *read*. If you can't read them, you can't change them (or fix them) and then you might as well throw them away and start from scratch. Do whatever you can to make your programs readable. You (or whoever will someday inherit and have to work with your code) will be glad you did.

7.4. FreePascal's Non-Numeric Ordinal Types

Many of Pascal's most useful types fall into a category we call *ordinal types*. An ordinal type has a limited number of discrete values that exist in one completely-defined and ordered series. There is a "first" value and a "last" value, and there are Pascal functions to allow you to move from one value in an ordinal type to the next value, or to the previous value. Pascal's ordinal types include **Char** and **Boolean**. The whole-number numeric types like **Integer**, **Byte**, **SmallInt**, **ShortInt**, **LongInt** and **LongWord** are ordinal types, but the 64-bit numeric types **Int64** and **QWord** are not. Enumerated types are also considered ordinal, but I'll treat them separately in Section 8.2.

Characters

The best way to explain Pascal's ordinal types is through a close look at the most common such type: **Char**. Type **Char** (character) is a Standard Pascal type, present in all implementations of Pascal. Type **Char** includes the familiar ASCII character set: Letters, numbers, common symbols, and the control characters like carriage return, backspace, tab, etc. There are 128 characters in the ASCII character set. But type **Char** actually includes 256 different values, since a character is expressed as an eight-bit byte. (Eight bits may encode 256 different values.) The "other" 128 characters have no standard names or meanings in the ASCII character set. When displayed on a device that supports their glyphs, the "high" 128 characters show up as non-English characters, fractions, segments of boxes, or mathematical and other symbols.

How, then, to represent such characters in your program? The key lies in the concept of *ordinality*. There are 256 different characters included in type **Char**. These characters exist in a specific ordered sequence numbered 0,1,2,3 and onward up to 255. The 65th character (counting from 0, remember) is always capital A. The 32nd character is always a space, and so on.

An ordinal number is a number indicating a position in an ordered series. A character's position in the sequence of type **Char** is its *ordinality*. The ordinality of capital A is 65. The ordinality of capital B is 66, and so on. Any character in type **Char** can be expressed by its ordinality, using the standard "transfer function" **Chr**. A capital A may be expressed as the character literal **'A'**, or as **Chr(65)**. The expression **Chr(65)** may be used anywhere you would use the character literal **'A'**.

Beyond the limits of the displayable ASCII character set, the **Chr** function is the only reasonable way to express a character. The character expressed as **Chr(234)** will display on the PC-compatible screen as the Greek capital letter omega (Ω) but may be displayed as some other glyph on another computer that is not PC-compatible. It is best to express such characters using the function **Chr**.

What will Pascal allow you to do with variables of type **Char**?

1. You can write them to the console display or printer using **Write** and **Writeln**:

```
Writeln('A');
Write(Chr(234));
Write(UnitChar);    { UnitChar is a variable of type Char }
```

2. You can concatenate them with string variables using the string concatenation operator (**+**) or the **Concat** built-in function. (See Section 12.2):

```
ErrorString := Concat('Disk error on drive ',UnitChar);
DriveSpec   := UnitChar + ':' + FileName;
```

3. You can derive the ordinality of characters with the **Ord** transfer function:

```
BounceValue := 31+Ord(UnitChar);
```

Ord returns a numeric value giving the ordinality of the character parameter. **Ord** allows you to perform arithmetic operations on the ordinality of a character. I'll discuss **Ord** (and its opposite number, **Chr**) in more detail in Section 11.5.

4. You can compare characters to one another with relational operators like =, >, <, >=, <=, and <>. (See Chapter 6.) This is due to the way characters are ordered in a series. What you are actually comparing is the ordinality of the two characters in their series when you use relational operators. For example, when you see this expression:

```
'a' > 'A'
```

(which evaluates to a boolean value of **True**) the computer is actually performing a comparison of the ordinalities of 'a' and 'A':

```
97 > 65
```

Since lower-case 'a' is positioned *after* upper-case 'A' in the series of characters, its ordinality is larger, and therefore 'a' is in fact "greater than" 'A', as peculiar as it may sound at first.

7.5. Booleans

Type **Boolean** is part of ISO Standard Pascal. A Boolean variable has only two possible values, **True** and **False**. Like type **Char**, type **Boolean** is an ordinal type, which means it has a fixed number of possible values that exist in a definite order. In this order, **False** comes before **True**. By using the transfer function **Ord** you would find that:

Ord(False) returns the value 0.

Ord(True) returns the value 1.

The status of the words **True** and **False** is a little tricky. In older versions of Pascal, including both the Borland Pascals and Delphi, **True** and **False** are *predefined identifiers* with no special status beyond that. The compiler predefines them as constants of type **Boolean**. This means that if you really want to, you can give **True** and **False** some other definition in your programs, as constants or variables. (This is a *very* bad idea and I don't recommend it!) When FreePascal is operating in Turbo mode ($MODE TP) or Delphi mode ($MODE DELPHI) this remains the case. However, in FreePascal's native mode ($MODE FPC) **True** and **False** are reserved words and may *not* be redefined. Unless you explicitly specify one of the other modes, FreePascal operates in FreePascal mode, so by default **True** and **False** are reserved words. (Because **True** and **False** have not always been reserved words in Pascal, I'm not placing them in uppercase as I do with other reserved words in this book.)

A Boolean variable occupies only a single byte in memory. The actual words **True** and **False** are not physically present in a Boolean variable. When a Boolean variable contains the value **True**, it actually contains the binary number 01. When a Boolean variable contains the value **False**, it actually contains the binary number 00. If you write a Boolean variable to a disk file, the binary values 00 or 01 will be physically written to the disk. However, when you print or display a Boolean variable using **Write** or **Writeln**, the binary values are recognized by program code and the words "**TRUE**" or "**FALSE**" (in uppercase ASCII characters) will be substituted for the binary values 00 and 01.

Boolean variables are used to store the results of expressions using the relational operators =, >, <, <>, >=, and <=, and the set operators +, *****, -. Operators and expressions will be discussed more fully in Section X.) An expression such as "**2 < 3**" is easy enough to evaluate; logically you would say that the statement "two is less than three" is "true." If this were put as an expression in Pascal, the expression would return a Boolean value of **True**, which could be assigned to a Boolean variable and saved for later processing:

```
OK := 2 < 3;
```

This assignment statement stores a Boolean value of **True** into the Boolean variable **OK**. The value of **OK** can later be tested with an **IF..THEN..ELSE** statement, with different actions taken by the code depending on the value assigned to **Ok**:

```
Ok := 2 < 3;
IF Ok THEN
  Writeln('>>Two comes before three, not after!')
ELSE
  Writeln('>>We are all in very serious trouble...');
```

Boolean variables are also used to alter the flow of program control in the **WHILE..DO** statement and the **REPEAT..UNTIL** statement. (See Sections 9.5 and 9.6.)

7.6. Integer Types in Detail

FreePascal adds considerable richness to the Standard Pascal suite of numeric types. Integer types **Byte**, **ShortInt**, **SmallInt**, **Word**, **LongInt**, **LongWord**, **Int64**, and **QWord**, and real types **Single**, **Double**, **Extended**, **Currency**, and **Comp** have come down to us via Turbo Pascal and Delphi, and did not exist in the original Pascal definition. With all that power come a few problems, and certainly a lot more details to remember as you design, write, and debug your FreePascal code.

Byte

Numeric type **Byte** is *not* present in Standard Pascal, although most microcomputer implementations of Pascal now include it. Type **Byte** may be thought of as an unsigned "half-precision" integer. It may express numeric values from 0 to 255. Like **Char**, **Byte** is stored in memory as an eight-bit byte. On the lowest machine level, therefore, **Byte** and **Char** are exactly the same. They only differ in what the compiler will allow you to do with them.

Byte variables may not share an assignment statement with any type that is not an integer type. Assigning a variable of type **Byte** with a variable or constant of any of the real number types, or with **Boolean**, **Char**, or any other non-numeric type will be flagged with a compile-time error. Type **Byte** may be freely included in expressions with the other numeric types described in this section. Type **Byte** may not, however, be assigned a numeric value of type **Real**, **Single**, **Double**, **Extended**, **Currency**, or **Comp**.

Range errors

There is machinery inside FreePascal's runtime library code to check whether the value assigned to a variable is suitable for that variable. With numeric values, this is known as *range checking*. If you attempt to assign a value to a numeric value and the value is out of range for that variable, FreePascal's runtime code can pop up an error message.

Range checking is possible both at compile time and at runtime. While the compiler is building your executable program, it can spot certain obvious range errors. For example:

```
VAR
  ByteItem : Byte

ByteItem := -17
```

Variables of type **Byte** may *not* taken on negative values. If you try to assign a negative constant to a variable of type **Byte**, you will get this message during the compile process:

```
Warning: range check error while evaluating constants
```

This is not a runtime error, because FreePascal can tell during compilation that the constant is negative, and it knows that type **Byte** cannot take a negative value. under any circumstances. Now, assigning a "signed" variable (like **Integer**, **ShortInt**, or **LongInt**) to **Byte** is perfectly safe unless (a) the signed variable contains a negative value, or (b) the value in the signed variable is too large to fit in a value of type **Byte**.

The sorts of things that the compiler can spot at compile time tend to be few and obvious. The truly gnarly problems will occur at runtime. Once your program hits the silicon and begins to crunch, runtime range checking becomes important.

Note well: *Runtime range checking is off by default.* You have to explicitly turn it on to use it. This is done by placing a compiler directive at the beginning of your program. The compiler directive looks like a comment, and actually is a comment of a special sort:

```
{$ RANGECHECKS ON}
```

The dollar symbol tells the compiler that this particular comment is intended for the compiler and not human programmers. Compiler directives are not actual Pascal code and do not need to be followed by a semicolon. They're instructions to the compiler telling it how to generate the code for your program. There are a fair number of them, though only a few will be useful to you while you're learning Pascal.

Below is a short program that commits a runtime range error. The **$RANGECHECKS** compiler directive is on. Read the program and see if you can tell what's wrong before continuing with the text:

```
 1 PROGRAM runtimetest1;
 2
 3 {$RANGECHECKS ON}
 4
 5 VAR
 6   I : Integer;
 7   ByteItem : Byte;
 8
 9 BEGIN
10   I := -17;
11   ByteItem := I;
12   Writeln(ByteItem);
13   Readln;
14 END.
```

This time the FreePascal compiler will not detect the problem when you compile your program. Difficulties will appear at runtime; that is, when you actually run the program. Two things may happen, depending on whether you have enabled range checking during the compilation of the program. If you ran the program from within the Lazarus environment and range checking was *on,* a message box will pop up that looks something like Figure 7.1.

If range checking was *not* on when you compiled the program, the program, in essence, will punt. It will do its best to pour a signed value into an unsigned variable, and what ends up in the variable depends on the physical bit-pattern of the negative value. To put it mildly, such errors are unpredictable, and because they happen in statements that work perfectly well most of the time, they can take a *great* deal of time and head scratching to locate and fix.

In general, each numeric type has a defined range, and if you enable range checking by using the **{$RANGECHECKSON}** compiler directive, assigning a value outside that range to a variable will generate a runtime error. This applies to the other numeric types discussed below as well as for type **Byte**.

Figure 7.1. A runtime range error message box
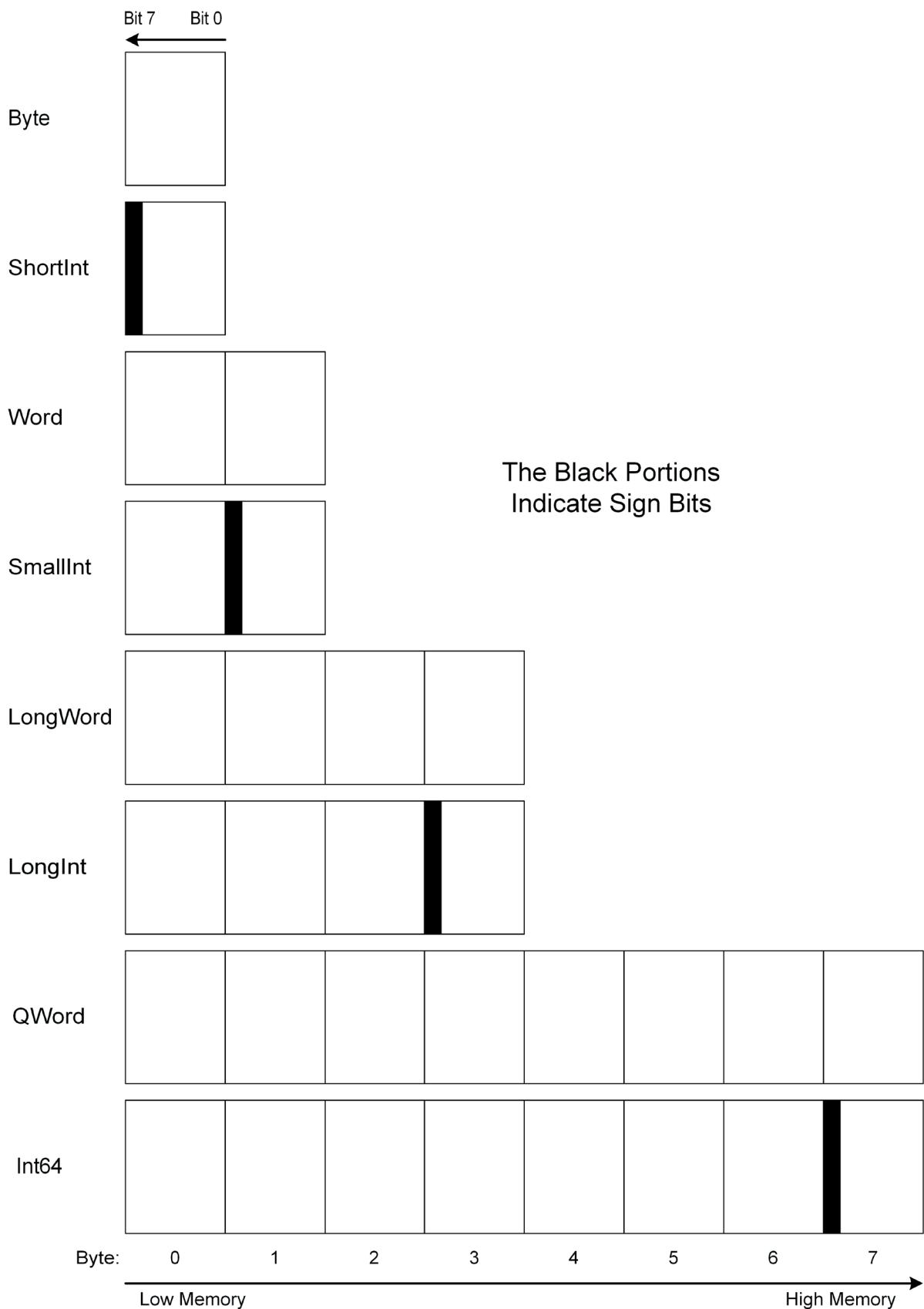
Figure 7.2. Memory Representation of Integer Types

Short integers

First cousin to type **Byte** is type **ShortInt**, a signed version of **Byte**. It may express values between -128 and 127. The problems of range errors exist for **ShortInt** just as they do for type **Byte**. **ShortInt** exists to provide a little bit of storage efficiency to programs that use a lot of small, signed values. If you use a lot of numeric variables, or (especially) large arrays of numeric variables, and can be sure the values will never wander out of the range -128..127, you can save a lot of space by using **ShortInt** variables instead of type **Integer**.

Bit 7 of a **ShortInt** is the sign bit. (See Figure 7.2.) If this bit is set to 1, the value is considered to be negative.

There is no significant speed improvement to be had by using **ShortInt** over larger numeric types, however. You might intuitively think physically small variables would be operated on more quickly than larger ones, but this is not necessarily the case. In fact, it takes post-8088 x86 CPU chips *longer* to process single-byte quantities in **Byte** and **ShortInt** variables than it does to process 16-bit or even 32-bit quantities. This is because Intel CPUs from the 286 back fetch and process data in 16-bit chunks, while chips from the 386 on fetch and process data in 32-bit and even (on 64-bit CPUs) 64-bit chunks. The newer CPUs must "stoop and grab" much more often when data exists in 8-bit chunks, and hence take longer to do repetitive operations. Use **Byte** and **ShortInt** to save space, not time!

Integer, LongInt, and SmallInt

Type **Integer** is a part of Standard Pascal. As a general rule, Pascal compilers make type **Integer** the size of the CPU accumulator register, since that is where whole-number math is done. In 8- and 16-bit CPUs, the accumulator register is 16 bits in size. In the 8-bit and 16-bit CPU era, type **Integer** was therefore two bytes in size and expressed a range of values from -32768..32767. With the advent of the 386, CPU register size jumped to 32 bits. Type **Integer** grew along with the accumulator register, and may now express values in the range -2147483648..2147483647.

Alas, even with 64-bit CPUs, type **Integer** remains 32 bits in size. This may change in the future. It's still true at this writing, in 2014.

Turbo Pascal 4.0 introduced the **LongInt** type to the Pascal language. **LongInt** was 32 bits in size, and allowed work on much larger whole-number values back when **Integer** was only 16 bits in size. Today, 16-bit CPUs are all but extinct in mainstream personal computing. Except for compiler versions targeting some very old or odd CPUs, types **Integer** and **LongInt** are now *precisely* the same thing. I recommend using **Integer** instead of **LongInt** whenever you can. If there's a reason for explicitly using **LongInt**, I suspect you'll know it.

Now, there are times when a 16-bit integer type is useful, especially when dealing with legacy code that makes assumptions about the underlying sizes of numeric types. This is a bad idea, but it was done a great deal in the past. (I wince a little when I remember doing it myself.) To serve this need, FreePascal offers the **SmallInt** type, which is a signed, 16-bit integer that can express values in the range -32768..32767. Be careful not to confuse the **ShortInt** and **SmallInt** types, which is easy enough to do when you're just starting out.

Sign bits and representation of integer types in memory

All Pascal data types are basically mapped to regions of machine memory. What parts of a data type correspond to what bits in memory is interesting to know, if not essential when you're a beginner. Later on it can be important, especially when you're writing programs that import and process data from some outside source, especially older data files.

The highest-order bit of the byte highest in memory is the *sign bit.* which indicates whether the value expressed by the integer is positive or negative. If this high bit is a binary 1, then the integer has a negative sign. If the high bit is a binary 0, the integer is positive. This is shown in Figure 7.2, where the dark bars represent sign bits. Where you don't see a dark bar, it means that the type shown is unsigned and does not have a sign bit.

Let me reiterate: I drew this figure for reference. You do *not* need to know which bit in a signed value is the sign bit except in very uncommon (and pretty advanced) coding circumstances.

Unsigned integer types

As you can see in Figure 7.2, half of FreePascal's integer types have no sign bit at all. Types **Byte**, **Word**, **LongWord**, and **QWord** are unsigned, and cannot express negative numbers. I'll give you a broad rule-of-thumb that reflects my 35+ years as a programmer: You will need signed integer values perhaps 20% of the time. Much depends on the sorts of programming you do. Scientific programming abounds in negative values. System programming and database programming generally do not. If you know your application will never encounter a negative value in certain parts of your code, you can buy additional positive range by using an unsigned integer type.

Of course, if you're wrong and your program attempts to assign a negative value to a variable of an unsigned integer type, your code will throw a runtime error.

Hi and Lo

There are circumstances where you may need to "pull apart" an integer type and inspect portions of it individually. This is done using a pair of built-in functions: **Hi** and **Lo**. The two functions essentially cut the memory space of a numeric value in half, and return either the high or the low half. "High" and "low" here refer to memory addresses, *not* values. As you might expect, **Hi** returns the higher half of a value's bytes as they reside in memory, and **Lo** returns the lower.

Don't confuse this with "cutting a value in half" in the sense of dividing it by two. Look back to Figure 7.2. If you call the **Hi** function on a variable of type **Word**, the function will return the higher of the value's two bytes in memory. If you call **Hi** on a variable of type **LongWord**, the function will return the variable's two bytes that are highest in memory. If you call **Hi** on a variable of type **QWord**, the function will return the value's four bytes that lie the highest in memory. The **Lo** function does the same thing, only returning the half of a value that lies lowest in memory.

A quick example: Given the 16-bit integer value 17,353 (hexadecimal equivalent $43C9), **Hi(17353)** will return 67 (hex $43) and **Lo(17353)** will return 201 (hex $C9). I include the hexadecimal equivalents because **Hi** and **Lo** are most typically used in system programming, much of which is documented and discussed using hexadecimal values.

Note that in using **Hi** and **Lo** on signed values, the sign bit is treated as just another bit in the high byte returned by **Hi**, and will not cause the value returned by either **Hi** or **Lo** to be returned as a negative quantity. For example, given the negative integer constant -21,244 (hex $AD04), **Hi(-21244)** will return 173 (hex $AD), and **Lo(-21244)** will return 4 (hex $04). **Hi** and **Lo** *never* return negative values.

If you're a newcomer and still a little fuzzy on the notion of Pascal functions, I'll be covering them in detail in Section 10.1.

7.7. REAL NUMBER VALUES AND TYPES

All the data types described up to this point have been ordinal types. Ordinal types are types with a limited number of possible values, existing in a definite order. Type **Char** is an ordinal type, as is **Boolean**, and all enumerated types. (See Section 8.2 for more on enumerated types.)

If an ordinal type directly expresses a numeric value, it is called a *scalar type*. Here's an example: Type **SmallInt** is an ordinal type, since it can express exactly 65,536 integer values. They are ordered and sharply defined: After 6 comes 7, after

7 comes 8, and so on, with no possible values in between. **SmallInt** is also a scalar type, since its values are numeric values, and not some other symbolic constants encoded internally as numeric values, as are **Boolean**'s **True** and **False**. All scalars are ordinals, but only ordinal types expressing numeric values are scalars.

Scalar types have absolute precision; that is, the value of the integer 6 is *exactly* six. (I have a marvelously ironic button reading, "2+2 = 5...for large values of 2.")

Computing with values obtained from the real world demands a way to deal with fractions. So Standard Pascal supports type **Real**, which can express numbers with fractions and exponents. Numbers like this are known as *rational numbers* in mathematics. In computing they are more often called *real numbers,* and are directly expressed in type **Real**. Real numbers, especially very large ones or very small ones, do *not* have absolute precision. For example, the scientific notation value 1.6125×10^{10} is expressed in Pascal as **1.6125E10.** The value is a real number having an exponent. You might expand the exponent and write it as 16,125,000,000. 1 This notation implies that we know the value precisely. However, we do not. A real number offers a fixed number of *significant figures* and an exponent giving us an order of magnitude, but there is a certain amount of "fuzz" in the actual value. The digits after the 5 in 16,125,000,000 are zeroes because we don't know what they really are. The measurements that produced the number were not precise enough to pin down the last six digits—so they are left as zeroes to express the order of magnitude that is expressed by the exponential in the form **1.6125E10**.

Real number types cannot be scalar types due to this lack of absolute precision. They are "real" in that they are usually used in the scientific and engineering community to represent physical measurements made of things in the real world. Integer types, by contrast, are largely mathematical in nature, and express abstract values usually generated by logic or calculation and not by physical measurement out here in the real world.

Real numbers may be expressed two ways in FreePascal. One way, as we've seen, is with an adaptation of scientific notation: a mantissa (in the example above, 1.6125) giving the significant figures, and an exponent (E10) giving the order of magnitude. This form is used for very large and very small numbers. For very small numbers, the exponent would be negative: **1.6125E-10**. You would read this number as "One point six one two five times ten to the negative tenth."

The second and more familiar way to express a real number is with a decimal point: 121.402, 3.14159, 0.0056, -16.6, and so on. Because using long strings of zeroes is inconvenient and an invitation to error (Count the zeroes!) decimal notation like this is best used for relatively small numbers.

You'll see the term "floating point" a lot in programming work, and it refers to the fact that the decimal point in a real number is not required to be at any particular place within however many significant figures a real number might have. For example, in a floating-point value with the six significant figures of accuracy 123456, the decimal point can be anywhere: 1.23456, 12.3456, 123.456, 1234.56, or 12345.6.

If this still isn't completely clear, compare these floating point examples to the **Currency** type, which is a "fixed-point" real, in which the decimal point is always placed four significant figures from the low end. This allows the expression of financial values accurate to a tenth of a mill. (A mill is one thousandth of a cent, and not often mentioned outside of financial circles.) More on **Currency** later.

FreePascal's real number types in detail

The original Turbo Pascal defined its own idiosyncratic type **Real**, consisting of a six-byte value having the range $10^{-38}..10^{38}$ with eleven significant figures. This was a reasonable thing to do in 1983, when math coprocessors barely existed and the CPUs of the time were not powerful enough to crunch the vast numeric ranges that we take for granted today.

This six-byte type **Real** is no longer used in FreePascal. Instead, FreePascal implements a suite of real-number types that conform to the IEEE floating-point specification, a well-established standard way of expressing floating-point values in computer memory. There was a time when using the IEEE real number types required the presence of a separate math coprocessor chip like the venerable 8087, and programs had to test for the presence of the chip. Beginning with the 486 family of Intel CPUs, a math coprocessor was integrated with the main CPU logic, so the IEEE real number types were always there, baked right into the silicon. On other CPUs for which FreePascal has been implemented, the IEEE floating-point types are absent and must be emulated, and special compile-time considerations may apply. I'm assuming Intel CPUs in this book, and will not discuss IEEE emulation any further. You'll find more about this arcane topic in FreePascal's documentation.

The IEEE real-number types are **Single**, **Double**, **Extended**, **Comp**, and **Currency**. They differ considerably in size and range:

- **Single** is a "single-precision" real number type. It is implemented in four bytes, and has a range of 10^{-38} to 10^{38}. This is the same *range* as the six-byte **Real** type, but because **Single** is implemented as only four bytes instead of six, it has less *precision* and will only yield 6 or 7 significant figures.

- **Double** is a "double precision" real number type. **Double** has a range of 5.0×10^{-324} to 1.7×10^{308}, with 16 significant figures of accuracy.

- **Extended** implements a "temporary real." It is expressed as ten bytes in memory and has the astonishing range of 10^{-4932} to 10^{4932} with 19 significant figures of accuracy.

- **Comp** is something of an outlier. It's lumped in with the real-number types, and yet it cannot take a decimal point and thus express a rational number. It's used as a signed integer with the range $-9.2 \times 10^{18}..9.2 \times 10^{18}$. In fact, **Comp** is bit-for-bit identical to the newer type **Int64**. However, for various peculiar historical reasons, **Comp** is treated differently by the FreePascal runtime library. (More on this a little later.) It's an obsolete type now and I suggest not using it at all. Use **Int64** (which is a true integer) or the **Currency** type for money values, keeping in mind **Currency**'s limitations.

- **Currency** is a type created to express money values with precision to one ten-thousandth of a cent. It can hold values up to a very respectable $922,337,203,685,477.5807 (nine-hundred twenty-two trillion dollars!) with complete precision down to a tenth of a mill. **Currency** shares some of **Comp**'s problems: On Intel x86 or x64 CPUs, both are considered floating-point types that are handled by the floating-point unit (FPU) and in consequence, there are some peculiar problems with rounding errors. On CPUs without an Intel FPU, **Currency**, like **Comp**, is mapped to **Int64**. **Currency** gets some special handling by the FreePascal runtime in terms of display and conversion to other types.

I should emphasize, of course, that nothing in the FreePascal compiler or the runtime libraries ties the **Currency** type to American dollars. Any decimal currency can be represented in variables of type **Currency**.

Extended's reason for existence

Extended is the most accurate and widest-ranging of any numeric type understood by the IEEE standard as implemented in Intel CPUs, and there are subtle dangers involved in using it. The term "temporary" as used in Intel's doc is quite apt: Whether or not you explicitly use type **Extended** in your programs, the math machinery has the type available to temporarily store intermediate values that might not be fully expressible in type **Double**. If you do a lot of calculations using enormous values in variables of type **Extended**, *the math subsystem no longer has a larger type to use to tuck away intermediate values.* In other words, if during a calculation an intermediate result appears that is larger that 10^{4932}, the math coprocessor simply doesn't have any way to express it, and a numeric overflow occurs. Your calculation will be inaccurate in both range or precision or both, and may trigger a runtime error. Where your value is over 1.0×10^{4932}, an *overflow error* occurs.

It strikes newcomers as odd sometimes, but you can generate an error by attempting to express a value in a floating point format that is too *small*. There are only so many orders of magnitudes to be had, even in the **Extended** type, and if your calculations yield a value that is smaller than 1.0 X 10^{-4932}, an *underflow error* will occur.

Intermediate results and subexpression promotion

This may be further understood in terms of what happens "behind the scenes" during calculations of very involved mathematical expressions. The FreePascal compiler evaluates expressions from left to right, "promoting" the type of the intermediate result to a larger numeric type as it goes. For example, consider this code snippet:

```
VAR
  RS : Single;
  RD : Double;

RS := 9.144E35; RD := 8.66543E255;
RS := ((RS*RS)*RD)/9.95E306;
```

The first portion of the expression to be evaluated is the subexpression **(RS*RS)**. Since the intermediate result generated by evaluating this subexpression has a magnitude 10^{70}, it is well beyond the range of type **Single**, which has a maximum positive magnitude of 10^{38}. Even though both variables in the subexpression are of type **Single**, the compiler "promotes" the intermediate value to type **Double**, whose maximum positive range of 10^{308} can comfortably handle it.

Having evaluated the subexpression **(RS*RS)**, the compiler moves to the right, and multiples that intermediate result by the value of **Double** variable **RD**, which at this point has a magnitude of 10^{255}. The new intermediate result has a magnitude of 10^{326}. This is beyond the 10^{308} expressible by type **Double**, so the compiler promotes the intermediate result to type **Extended**. Finally, the intermediate result is divided by a literal constant with a magnitude of 10^{306}. This reduces the magnitude of the intermediate result to 10^{20}, which is comfortably within the range of type **Single**.

Now, what would have happened had there not been a type **Extended** for the intermediate result to be promoted to during the evaluation of this expression? The code generated by FreePascal would have assigned the pseudo-value **INF** to the expression, and this pseudo-value would have carried through the evaluation and been assigned to **Single** variable **RS**, even though **RS** had enough range to accommodate the final result. And while **INF** is a defined and legal IEEE floating point value, it is a difficult thing to respond to in ordinary arithmetical calculations.

Note that the general principles of promoting intermediate results as required applies to integer as well as floating point expressions. The lesson in the previous example is that in expressions containing variables of type **Extended**, there is nothing larger to store intermediate results in case of an overflow to IEEE infinity. Use **Extended** with extreme care, not only in writing expression so as to minimize the "explosion" of intermediate results, but also in keeping an eye on the values taken on by **Extended** variables at runtime.

The challenge of dealing with enormous numbers in the thick of calculations is a longstanding issue in programming. FreePascal does a very good job, but you have to be aware of the practical limitations of today's CPUs.

The odd case of Comp

FreePascal's suite of IEEE-compatible types contains two types that are functionally integer types but are almost always lumped in with the various IEEE real number types as "computational reals." One is type **Comp**. It has a range of -9.2×10^{18} to 9.2×10^{18}. Its distinctive feature is that it has absolute precision throughout its range. **Comp** trades range for precision. It has far less range than **Single**, but there is no value at which it "fuzzes out" as conventional real-number values do.

Comp thus hovers between two worlds. While it's true that **Comp** doesn't take a decimal part as true floating point numbers do, it acts more like a floating point type than an integer type in several important ways. First of all, it does not work with integer division operators **MOD** and **DIV**. You must use the / operator to perform division on a variable of type **Comp**. Second, its default display format is the exponential format used by all other floating-point numbers. To display a **Comp** variable with neither exponential notation nor unused and unusable decimal places you must format it as you would a real number. For example:

```
VAR
  BigNum : Comp;

TestComp := 17284;
Writeln(TestComp);        { Displays as 1.72840000000000E+0004 }
Writeln(TestComp:7);      { Displays as 1.7E+0004 }
Writeln(TestComp:7:2);    { Displays as 17284.00 }
Writeln(TestComp:7:0);    { Displays as   17284 }
```

Currency, the money type

Type **Comp** goes back a long way. It was a response to the need for doing money math on very large figures, with accuracy to a thousandth of a cent or more. In 1995, Delphi introduced a new and better—if not perfect—solution to the problem of money math. This is the **Currency** type.

Currency is related to **Comp**, and the two types both occupy 8 bytes in memory. **Currency** has the same precision as **Comp**, but its range is less, because **Currency** values are assumed to have a decimal point and four places of precision after the decimal. In a sense, **Currency** is a "fixed-point" real number with complete precision throughout its range, which needs a whole line to state completely:

-$922,337,203,685,477.5808 to $922,337,203,685,477.5807

This is just short of a quadrillion dollars, which should keep financial analysts happy for at least a little while. We hope.

Currency's issues

As good as **Currency** sounds, there are some issues in using it. As with **Comp**, on CPUs containing an Intel x87 FPU **Currency** is handled by the FPU. (On CPUs without an x87 FPU, **Currency** cooks down to type **Int64**.) It's hard to explain in detail in an introductory book like this, but there are conversion problems when converting values stored in a **Currency** variable to a value stored in one of the floating-point types. It's best to keep values in the **Currency** type as much as possible during your calculations. In particular, do *not* attempt to use the absolute value function **Abs** with **Currency**. Behind the scenes, **Abs** converts the **Currency** value to **Extended** and then back again, with rounding errors as an unintended consequence. (This occurs only on CPUs containing an x87 FPU.)

The other issue is a difference in rounding methods between Delphi and FreePascal. There are two types of rounding in financial calculation. One is the method most of us learned in grade school, in which a value like 65.285 would be rounded up to 65.29. The other is "bankers' rounding," in which numbers like 65.285 would be rounded to the closest even number, which in this example would be 65.28. (Thanks to Mike Riley for pointing this out to me.)

The Delphi runtime library uses bankers' rounding for **Currency** values, whereas FreePascal's implementation of **Currency** uses conventional rounding. If you're bringing Delphi code over to Lazarus/FreePascal, this may become a problem.

Currency, like **Comp**, has special issues with conversion to text for display. I'll deal with that matter later in this book.

CHAPTER 8.
DERIVED TYPES AND
DATA STRUCTURES

All the types we've discussed up to this point have been simple types, predefined by FreePascal and ready to use. Much of the power of Pascal lies in its ability to create more complex structures of data out of these simple types. Derived types and data structures can make your programs both easier to write and, later on, easier to read as well.

We touched on this earlier in the book. Recapping: Creating custom data types is easy to do. The reserved word **TYPE** begins the type definition part of your program, and that's where you lay out the plan of your data structures:

```
TYPE
  YourType = ItsDefinition;
```

In general terms, a type definition consists of the name of the type, followed by an equal sign, followed by the definition of the type. From now on, many of the types we're going to discuss must be declared and defined in the type definition part of your program. Once defined, you can declare a variable of your "custom" type in the variable declaration section of your program:

```
VAR
  ANewVariable : YourType;
```

A custom type of this sort is often called a *derived type* in Pascal circles.

A type *definition* does not, by itself, occupy space in your executable program file's data area, as variables do. What a type definition provides are instructions *to the compiler* telling it how to deal with variables of type **YourType** when it encounters them further down in your program source file.

With some few exceptions, (strings and subranges, for example) you cannot write derived or structured types to your display or printer with **Write** or **Writeln**. If you want to display derived or structured types somehow, you must write procedures specifically to display some representation of the type on your display or printer.

8.1. OVERVIEW: TYPES AS BRICKS TO BUILD WITH

Virtually any task can be accomplished using Pascal's fundamental data types alone. However, creating structures of data by using these fundamental data types as building blocks can help you develop a program design, and help you code the program once the design is complete. Using its basic data types, a Pascal programmer can build special-purpose types that are valid only within the program in which they are defined.

One way to build new special-purpose data types is by defining subranges. A *subrange* is a Pascal type that may have as its values only certain values in a range taken from the legal range of the fundamental data type. For example, academic grades are usually expressed as letters. Not all letters are grades, however. (Did you ever get a "W" in history?) You might define a subrange of the basic type **Char** that can have as values only the letters from 'A' through 'F'. Such a subrange type would be defined this way:

```
Grade = A..F;
```

Now, to create a variable to hold grades, you would declare a variable this way:

```
History : Grade;
```

Subranges provide a modicum of protection against certain coding mistakes. For example, if you tried to assign a grade of "W" to the variable **History**, the compiler would tell you during compilation that "W" is not a legal value of the subrange **Grade**. At runtime, the compiler would not be available for comment. However, if you tried to assign a **Char** variable containing "W" to a **Grade** variable at runtime, you would generate a runtime error—*if* range checking were enabled.

Subranges are derived types. They're subsets of existing types, either those built into Pascal or those you define yourself. Building the other way happens when you create data structures, the most familiar of which is called a *record*. A record is a grouping of existing types into a larger structure that is given a name as a new type. Variables can be declared to be of this type. Those variables can be assigned, compared, and written to files just as integers or characters can.

For example, if you were writing a program that keeps track of student grades and test results, you would first define separate variables for all the various data items that express a student's individual grades for a given semester. Some of these variables might be basic Pascal types like **Integer**, and others might be derived data types, like the **Grade** type we defined earlier. You might then group together the separate variables as a record type, to express a student's status for an entire semester. Such a record type might look like this:

```
TYPE
  SemesterGrades = RECORD
                    StudentID   : String[9];
                    SemesterID  : String[6];
                    Math        : Grade;
                    English     : Grade;
                    Drafting    : Grade;
                    History     : Grade;
                    Spanish     : Grade;
                    Gym         : Grade;
                    SemesterGPA : Real
                  END;
```

Now you can define a Pascal variable as having the type **SemesterGrades:**

```
VAR
  ThisSemester : SemesterGrades;
```

By a single variable name you now can control nine separate data chunks (which, when part of a record type, are called *fields*) that you would otherwise have to deal with separately. This can make certain programming tasks a *great* deal simpler. But even more important, it allows you to treat logically-connected data as a single unit to clarify your program's design and foster clear thinking about its function.

For example, when you need to write the semester's grades out to a disk file, you needn't fuss with individual subjects separately. The whole record goes out to disk at once, as though it were a single variable, without any reference to the individual fields from which the record is built. The alternative is a series of statements that write the student ID to disk, followed by the semester ID, followed by the math grade, followed by the English grade, and so on.

One caution about this particular example: If you've ever worked with a database manager, storing Pascal records in a Pascal file is very close to what a database manager does when it writes individual database records to a database table. FreePascal can work very effectively with external database engines like MySQL or SQLite, so writing records out as simple files is not done much anymore. You simply need to be aware that it can be done. Data management these days requires a "real" database engine. You don't have to write an application-specific database manager yourself!

Pascal records really aren't about writing groups of variables to disk. It's about how you think about the problem at hand: When you need to think of all of a student's grades taken together, you can think of them as a unit, in the form of a record. When you need to deal with them separately, Pascal has a simple way of picking out any individual field within the record for individual attention. Selecting one field out of a record is done this way:

```
MyMath := ThisSemester.Math;
```

You simply specify the record name followed by the field name, separated by a period character. **ThisSemester.Math** is in a sense an expression that "cooks down" to a single value of type **Grade**. Informally, this is sometimes called "dotting."

How you think of the data now depends on how you *need* to think of the data. Pascal encourages you to structure your data in ways like this that encourage clear thinking about your problem at a high level (all grades taken together) or at a low level (each grade a separate data item.)

Much of the skill of programming in Pascal is learning how to structure your data so that details are hidden by the structure until they are needed. It's much like being able to step back and see your data as a forest without being distracted by the individual trees.

Like most tools, the structuring of data is an edge that cuts two ways. It is all too easy to create data structures of Byzantine complexity that add nothing to a program's usefulness while obscuring its ultimate purpose. If the data structure you create for your program makes the program *harder* to understand from "three steps back," you've either done it the wrong way, or done it too much.

The rule of thumb I use is this: *Don't create data structures for data structure's sake.* Unless there's a reason for it, resist. Simplicity doesn't necessarily sacrifice power or flexibility.

Subranges in Detail

If you choose any two legal values in an ordinal type, those two values plus all values that lie between them define a subrange of that ordinal type. For example, these are subranges of type **Char**:

```
TYPE
  Uppercase = 'A'..'Z';
  Lowercase = 'a'..'z';
  Digits    = '0'..'9';
```

Uppercase is the range of characters A,B,C,D,E,F and so on to Z. **Digits** includes the numeral characters 0,1,2,3,4 on to 9. The quotes are important. They tell the compiler that the values in the subrange are of type **Char**. If you were to leave off the quote marks from the type definition for type **Digits**:

```
Digits = 0..9;
```

you would have, instead, a subrange of type **Integer**. '7' is not the same as 7!

An expression in the form **'A'..'Z'** or **3..6** is called a *closed interval*. A closed interval is a range of ordinal values including the two stated boundary values and all values falling between them. We'll return to closed intervals later on in this chapter while discussing sets.

8.2. ENUMERATED TYPES

Newcomers to Pascal frequently find the notion of enumerated types hard to grasp. An enumerated type is an ordinal type defined by the programmer. It consists of an ordered list of values, where each value has a unique name. One of the best ways to approach enumerated types is through comparison with type **Boolean**.

Type **Boolean** is, in fact, an enumerated type that is predefined by the FreePascal compiler and used in special ways. Type **Boolean** is an ordered list of two values with unique names: **False**, **True**. It is *not* a pair of ASCII strings containing the English words "False" and "True." As we mentioned earlier, a Boolean value is actually a binary number with a value of either 00 or 01. We "name" the binary code 00 within type **Boolean** as **False**, and name the binary code 01 within type **Boolean** as **True**. It's a process similar in spirit to naming constants.

Consider another list of values with unique names: the colors of the spectrum. Let's create an enumerated type in which the list of values includes the colors of the spectrum, in order:

```
TYPE
  Spectrum = (Red, Orange, Yellow, Green, Blue, Indigo, Violet);
```

The values list of an enumerated type definition is always given within parentheses. The order you place the values within the parentheses defines their ordinal value, which you can test using the **Ord(X)** function. For example, **Ord(Yellow)** would return a value of 2. **Ord(Red)** would return the value 0. **Ord(Indigo)** would return the value 5.

You can compare values of an enumerated type with other values of that same type. It may be helpful to substitute the ordinal value of enumerated constants for the words that name them when thinking about such comparisons. The statement **Yellow > Red** (think: 2 > 0) would return a Boolean value of **True**. **Green > Violet** or **Blue < Orange** would both return Boolean values of **False**. Comparisons between enumerated types is about their position in a number line, *not* their names!

Individually, the values of type **Spectrum** are all considered named constants. They may be assigned to variables of type **Spectrum**. For example:

```
VAR
  Color1, Color2 : Spectrum;

Color1 := Yellow;
Color2 := Indigo;
```

You cannot, however, assign just anything to one of the values of type **Spectrum**. Statements like **Red := 2** or **Red := Yellow** make no sense and will not be accepted by the compiler.

Enumerated types may index arrays. (I'll be discussing arrays in detail a little later in this chapter.) For example, each color of the spectrum has a frequency and wavelength associated with it. These frequencies could be stored in an array, indexed by the enumerated type **Spectrum**:

```
Wavelength : ARRAY[Red..Violet] OF Real;
Frequency  : ARRAY[Red..Violet] OF Real;
Color      : Spectrum;
Lightspeed : Real;


Wavelength[Red]    := 6.2E-7;    { All in meters }
Wavelength[Orange] := 5.9E-7;
Wavelength[Yellow] := 5.6E-7;
Wavelength[Green]  := 5.4E-7;
Wavelength[Blue]   := 5.15E-7;
Wavelength[Indigo] := 4.8E-7;
Wavelength[Violet] := 4.5E-7;
```

The functions **Ord** and **Odd** work with enumerated types, as do the **Succ** and **Pred** functions. This is due to an enumerated type's having a fixed number of elements in a definite order that does not change. **Succ(Green)** will always return the value **Blue**. **Pred(Yellow)** always returns the value **Orange**. Be aware that **Pred(Red)** and **Succ(Violet)** are undefined. There is no value before **Red** or after **Violet**. You should test for the two ends of the **Spectrum** type while using **Succ** and **Pred** to avoid assigning an undefined value to a variable.

Enumerated types may also be control variables in **FOR/NEXT** loops. (These will be discussed in detail in the next chapter.) In continuing with the example begun above, we might calculate the frequencies of light for each of the colors of type **Spectrum** this way:

```
Lightspeed := 3.0E08;             { Meters/second }
FOR Color := Red TO Violet DO
  Frequency[Color] := Lightspeed / Wavelength[Color];
```

Displaying or printing enumerated type values

FreePascal differs from most earlier Pascal compilers in that it will display individual values from an enumerated type, when those values are not, strictly speaking, text strings.

For example, if you write the value **Orange** (from type **Spectrum**) to the console like this:

```
Writeln(Orange);
```

your program will display the ASCII word "Orange" in the console window. This has not always been the case. Turbo Pascal used to flag this with the following error:

```
Error 64: Cannot Read or Write variables of this type
```

FreePascal, by contrast, is more than happy to display or print the enumerated type values as you defined them. There's some potential for confusion here, however: Internally, the values of enumerated types are maintained as binary numbers, *not* ASCII strings like "Green."

8.3. Sets and Set Operators

Sets are collections of elements picked from simple types. An element is either in a set, or it is not in the set. The letters "A", "Q", "W", and "Z" may be taken together as a set of characters. "Q" is in the set, and "L" is not.

Expressed in Pascal's notation:

```
VAR
  CharSet : SET OF Char;

CharSet := ['A','Q','W','Z'];
```

A pair of square brackets when used to define a set (as shown above) is called a *set constructor.*

Pascal sets are very useful, often in non-obvious ways. For example, sets provide an easy way to sift valid user responses from invalid ones. In answering even a simple, yes/no question, a user may in fact type two equally valid single characters for yes, and two for no: Y/y and N/n. Ordinarily, you would have to test for each one individually:

```
IF (Ch='Y') OR (Ch='y') THEN DoSomething;
```

With sets, you could replace this notation with:

```
IF Ch IN ['Y','y'] THEN DoSomething;
```

The operator **IN** tests whether the value stored by **Ch** is present in the set. **IN** returns **True** if an element is present in a set, or **False** if an element is not in the set.

Base types and closed intervals

In FreePascal, a set type may be defined for (almost) any simple type having 256 or fewer individual values. The type from which sets elements are derived is called that set's *base type*. Type **Char** qualifies as a base type, as does **Byte**. The enumerated type **Spectrum** we created in the last section also qualifies, since it has only seven separate values. A set of **Spectrum** might contain some but not all the colors defined as Spectrum's values:

```
VAR
  LowColors : SET OF Spectrum;

LowColors := [Red,Orange,Yellow];
```

Only one numeric type may be a set base type: **Byte**. The types **ShortInt** and larger numeric types do not qualify. Subranges may be the answer: If you define a numeric subrange spanning 256 or fewer values, you may define a set with that subrange type as the set's base type:

```
TYPE
  ShoeSizes   = 5..17;

VAR
  SizesInStock : SET OF ShoeSizes;
```

In addition to establishing the elements present in a set in the **VAR** and **TYPE** sections of your program, you may also assign a range of elements to a set in an assignment statement, assuming that the elements assigned are of an acceptable base type:

```
VAR
  Uppercase, Lowercase, Whitespace, Controls : SET OF Char;

Uppercase  := ['A'..'Z'];
Lowercase  := ['a'..'z'];
Controls   := [Chr(1)..Chr(31)];
Whitespace := [Chr(9),Chr(10),Chr(12),Chr(13),Chr(32)];
```

This is certainly easier than explicitly naming all the characters from A to Z to assign them to a set. A range of elements containing no gaps (like **'A'..'Z'**) is called a *closed*

interval. The list of members within the set constructor can include single elements, closed intervals, and expressions that yield an element of the base type of the set. These must all be separated by commas, but they do not have to be in any particular order:

```
X : Byte;


X        := 77;
GradeSet := ['A'..'F','a'..'f'];
BadChars := [Chr(1)..Chr(8),Chr(11),Chr(X+4),'Q','x'..'z'];
```

You should take care that expressions do not yield a value that is outside the range of the set's base type. If **X** in the **BadChars** type defined above grows to 252 or higher, the result of the expression **Chr(X+4)** will no longer be a legal character. The results of such an expression will be unpredictable, other than to say they won't do you very much good.

Sets like **Uppercase**, **Lowercase**, and **Whitespace** defined above can be very useful when manipulating characters typed at the keyboard or from another unpredictable source. Here are some simple functions making use of these character sets:

```
FUNCTION CapsLock(Ch : Char) : Char;

BEGIN
  IF Ch IN Lowercase THEN CapsLock := Chr(Ord(Ch)-32)
    ELSE CapsLock := Ch
END;


FUNCTION DownCase(Ch : Char) : Char;

BEGIN
  IF Ch IN Uppercase THEN DownCase := Chr(Ord(Ch)+32)
    ELSE DownCase := Ch
END;


FUNCTION IsWhite(Ch : Char) : Boolean;

BEGIN
  IsWhite := Ch IN WhiteSpace
END;
```

All three of these functions assume that **Uppercase**, **Lowercase**, and **Whitespace** have already been declared and filled with the proper values. Actually, the way to ensure that this is done is to do it *inside* each routine by the use of set constants, as I'll explain toward the end of this chapter.

CapsLock returns all characters passed to it in uppercase. **DownCase** returns all characters passed to it as lowercase. **IsWhite** returns a **True** value if the character passed to it is "whitespace", that is, a tab, carriage return, linefeed, or space character.

A set may be defined as having no elements at all. This is called a *null set*. When you declare a set variable, that variable is initially a null (empty) set. If it is to contain any elements, you must add those elements in a separate statement. You can also assign a null set to a set variable that may have elements in it. This "empties out" the set, which is a useful operation all by itself:

```
VAR
  NullSet = SET OF Spectrum;

NullSet := [Green, Blue];
NullSet := [];            { 'Empty out' NullSet by assigning it a null set. }
```

Set operators

The **IN** operator we used above is not the only operator available for use with sets. There are two whole classes of set operators in FreePascal: Operators that build sets from other sets, and operators that test relationships between sets and return a Boolean result.

Pascal allows you to create new sets from existing sets using several operators. To a great extent the operators follow the rules of set arithmetic you may have learned in grade school. Table 8.1 summarizes the set operators implemented in FreePascal. All of them take set operands and return set values. Note that **Include** and **Exclude** are implemented as functions in FreePascal, so although they act like operators, they look like functions that return set types.

Table 8.1. Set-builder operators and functions

Operator	Symbol	Logic equivalence	Precedence
Set union	+	OR	4
Set intersection	*	AND	4
Set difference	-		4
Set symmetric difference	><	XOR	4
Set include (function)	Include	Union with 1 element	
Set exclude (function)	Exclude	Subtraction with 1 element	

Set union

The union of two sets is the set that contains all members contained in both sets. In simpler terms, it means combining the two sets into a single set. The symbol for the set union operator is the plus sign (**+**), just as for arithmetic addition. An example:

```
VAR
  SetA, SetB, SetX, SetY, SetZ : SET OF Char;

SetX := ['Y','y','M','m']; SetY := ['N','n','M','m'];
SetZ := SetX + SetY;
```

After the set union operation, **SetZ** contains '**Y**', '**y**', '**N**', '**n**', '**M**', and '**m**'. Note that although '**M**' and '**m**' exist in both **SetX** amd **SetY**, each appears only *once* in the union of the two sets. A set merely says whether or not a member is present in the set; it is meaningless to speak of how many *times* a member is present in a set. By definition, each member is present only once, or not present at all.

Set difference

The "difference" of two sets is conceptually related to arithmetic subtraction. Given two sets **SetA** and **SetB**, the difference between them (expressed as **SetA - SetB**) is the set consisting of the elements of **SetA** that remain once all the elements of **SetB** have been removed from it. For example:

```
SetA := ['Y','y','M','m'];
SetB := ['N','n','M','m'];
SetX := SetA - SetB;
```

SetX now contains **'Y', 'y'**. Conceptually, set difference "pulls out" whatever **SetB** contains that is also present in **SetA**. **SetB** does not have to be a subset of **SetA**. In other words, **SetB** may contain elements that are not present in **SetA**.

Set intersection

The intersection of two sets is the set that contains as members only those members contained in *both* sets. The symbol for set intersection is the asterisk (*****), just as for arithmetic multiplication. For example:

```
SetX := ['Y','y','M','m'];
SetY := ['N','n','M','m'];
SetZ := SetX * SetY;
```

SetZ now contains **'M'** and **'m'**, which are the only two members that are contained in both sets.

Set symmetric difference

FreePascal adds a set operator that, to my knowledge, does not exist in any version of Borland's Pascal products. This is the *symmetric difference* operator, and as with the others I've just described, it comes to us from set logic. The symbol for set symmetric difference is ><. The symmetric difference between two sets is whatever remains when the intersection of the two sets has been removed from the union of the two sets.

```
SetX := ['Y','y','M','m','C','c'];
SetY := ['N','n','C','c'];
SetZ := SetX >< SetY;
```

Here, the intersection of **SetX** and **SetY** is the set **['C','c']**. The union of the two sets is **['Y','y','M','m','N','n','C','c']**. If you remove the intersection of the two sets from the union of the two sets, what you have left (the symmetric difference) is **['Y','y','M','m','N','n']**.

Include and Exclude for sets

FreePascal has another bit of set machinery that was not present in any of Borland's Pascal compilers: *set include* and *set exclude*. The idea here is to add or remove a single element at a time. The two operations are implemented as Pascal functions rather than operators. This is simple enough, but it shows better than it tells. So let's return to the example of an enumerated type containing a sequence of color values. The **MyColors** set variable is assigned the list of legal colors in the **Spectrum** type:

```
TYPE
  Spectrum = (Red, Orange, Yellow, Green, Blue, Indigo, Violet);

VAR
  MyColors : SET of Spectrum; { MyColors is created as a null set!  }

MyColors := [Red, Orange, Yellow, Green, Blue, Indigo, Violet];
Exclude(MyColors,Green);      { Pulls Green out of the set MyColors }
```

A quick reminder for beginners: Declaring a variable as a set of an enumerated type does *not* give the set any values from the enumerated type! When you declare a set variable, that variable is a null set until you put something in it. Any values have to be assigned to the new set in a separate statement.

Here, the **Exclude** function pulls the value **Green** out of a set called **MyColors**. The result is the same as though you had created a null set and then added **Green** to it, followed by set subtraction:

```
VAR
  ColorToBeRemoved : SET of Spectrum;

ColorToBeRemoved := [Green];
Colors := Colors – ColorToBeRemoved;
```

Similarly, the **Include** function adds a single element to a set, assuming the element isn't in the set already:

```
VAR
  CharSet,JustR : SET OF Char;

CharSet := ['A','Q','W','Z'];
Include(Charset,'R');
```

After the **Include** function executes, **Charset** contains **['A','Q','R','W','Z']**. The results would be the same if you created a set with **'R'** in it, and then performed set addition on the two sets:

```
JustR   := ['R'];
Charset := Charset + JustR;
```

Table 8.2. Set relational operators.

Operator	Symbol	Precedence
Set equality	=	4
Set inequality	<>	4
Element inclusion	IN	4
Set inclusion, left in right	<=	4
Set inclusion, right in left	=>	4

The set relational operators

The set operators just described work with set operands to produce new set values. I've briefly mentioned the relational operators that test relationships between sets and return Boolean values depending on those relationships. In this section we'll take a more detailed look. The set relational operators are summarized in Table 8.2.

Sets can be compared in various ways. For example, sets can be equal to one another, if they both have the same base type and both contain the same elements. Two sets that are *not* of the same base type will generate an "incompatible types" error at compile time if you try to compare them:

```
VAR
  SetX : SET OF Char;
  SetQ : SET OF Spectrum;
  OK   : Boolean;

OK := SetX = SetQ;   { Will trigger "incompatible types" error! }
```

This holds true for *all* set relational operators, not just equality as demonstrated above.

The single most important set relational operator is the element inclusion operator **IN**. **IN** tests whether a value of a set's base type is a member of that set.

```
VAR
  Ch : Char;

Read(Ch);                    { From the keyboard }
IF Ch IN ['Y','y'] THEN
  Write('Yes indeed!');
```

As described earlier, this example provides a clean and easy way to tell whether a user has typed the letter Y (upper or lower case) in response to a prompt. The **IN** operator tests whether the typed-in character is a member of the set constant **['Y','y']**. (**IN** works just as well with set variables.)

The greater than (>) and less than (<) operators make no sense when applied to sets, because sets have no implied order to their values. However, there are two additional set relational operators that make use of the same symbols as used by the greater than or equal to (>=) and less than or equal to (<=) operators. These are the set inclusion operators.

Inclusion of left in right (<=) tests whether all members of the set on the left are included in the set on the right. Inclusion of right in left (>=) tests whether all members of the set on the right are included in the set on the left. The action these two operators take is identical except for the orientation of the two operands with respect to the operator symbol. Given two sets, **Set1** and **Set2**, this expression:

```
(Set1 <= Set2) = (Set2 >= Set1)
```

will always evaluate to **True**.

This may seem a little arcane. It may be better to think of set inclusion as a way to test whether one set is a subset of another. This is very handy for testing and manipulating characters in a text stream. For example:

```
VAR
  Vowels,Alphabet,Samples : SET OF Char;

Vowels:=['A','E','I','O','U'];
Alphabet:=['A'..'Z'];            { Set of all UC letters }
Samples:=['A','D','I','Q','Z'];

IF Samples <= Vowels THEN Write('All samples are vowels.');
IF NOT(Samples <= Vowels) AND (Samples <= Alphabet) THEN
  Write('Some or all samples are uppercase letters.');
IF NOT(Alphabet >= Samples) THEN
  Write('Some samples are not uppercase letters.');
```

In addition to demonstrating the set inclusion operators, these examples also show some uses of the **AND** and **NOT** logical operators in **IF** statements, which I'll explain in detail in the next chapter. Given the elements assigned to **Samples** in the above example, this output will be displayed:

```
Some or all samples are uppercase letters.
```

Now, as practice before going on, jot down two sets of characters that would trigger the other two messages in the above example.

Avoid set inclusion confusion

Don't get the set inclusion operators **<=** and **>=** mixed up with element inclusion operator **IN**. When you use **IN**, you're testing whether *one single value* of a set's base type is present in the set. The set inclusion operators test whether *all* the elements of one set are present in another set. The following expression may seem to make sense, but FreePascal won't allow it:

```
Vowels IN Alphabet
```

Both **Vowels** and **Alphabet** are sets of characters. To test whether **Vowels** is a subset of **Alphabet**, you need the left-in-right set inclusion operator:

```
Vowels >= Alphabet
```

This will compile, and simply tests whether everything in the **Vowels** set is also present in **Alphabet**. With the values given earlier, the expression evaluates to **True**.

Note: For space reasons I have chosen not to discuss the binary representation of sets in this book.

8.4. ARRAYS

Sometimes it's not enough to work on a single data item, whatever its type. Sometimes you need to work on a whole row of them. If all the data items in the row are all of the same type, you can refer to them by number, just as you might choose photograph #6 from an old roll of film, or the legendary Love Potion #9. Pascal can do that, using data structures called arrays.

An *array* is a data structure consisting of a fixed number of *elements* of the same type, with the whole data structure given a single identifier as its name. The program keeps track of individual elements by number. Sometimes you name the entire array to work with it as a unified whole. Most of the time you identify one of the individual elements, by number, and work with that element alone. The number identifying an array element is called an *index*. In Pascal, an index need not always be a traditional number. Enumerated types, characters, and subranges may also act as array indices.

It may be helpful to think of an array as a row of identical empty boxes in memory, side by side. The program allocates space for the boxes, but it is your job as programmer to fill them and manipulate their contents. The elements in an array are, in fact, set side-by-side in order in memory. (You don't need to know how arrays are laid out in memory in order to use them during ordinary Pascal programming.)

An array element may be of any data type except a file. Arrays may consist of data structures; that is, you may have arrays of records and arrays of arrays. An array index must be a member or a subrange of an ordinal type, or a programmer-defined enumerated type. Floating point numbers may *not* act as array indices, nor may **LongInt** or **Comp**. Type **Integer**, **ShortInt**, **Byte**, **Word**, **Char**, and **Boolean** may all index arrays.

Below are some valid array declarations, just to give you a flavor of what's possible. (If you don't understand for now what records or strings are, bear with me for the time being. They're up next!)

```
CONST
  Districts  = 14;


TYPE
  String80   = String[80];
  Grades     = 'A'..'F';                        { Subrange  }
  Percentile = 1..99;                           { Ditto     }
                                                { Enum. type }
  Levels     = (K,G1,G2,G3,G4,G5,G6,G7,G8,G9,G10,G11,G12);
                                                { Ditto     }
  Subjects   = (English,Math,Spelling,Reading,Art,Gym);
```

```
  Profile     = RECORD
                    Name     : String80;
                    SSID     : String80;
                    IQ       : Integer;
                    Standing : Percentile;
                    Finals   : ARRAY[Subjects] OF Grades
                END;

  GradeDef  = ARRAY[Grades] OF String80;


VAR
  K12Profile : ARRAY[Levels] OF Profile;
  Passed     : ARRAY[Levels] OF Boolean;
  Subtotals  : ARRAY[1..24] OF Integer;
  AreaPerc   : ARRAY[1..Districts] OF Percentile;
  AreaLevels : ARRAY[1..Districts] OF ARRAY[Levels] OF Percentile;
  RoomGrid   : ARRAY[1..3,Levels] OF Integer;
```

The declarations shown above are part of an imaginary school district records manager program written in Pascal. Note that **Passed** is an array whose index is an enumerated type. **Passed[G5]** would contain a Boolean value (**TRUE** or **FALSE**) indicating whether a student had passed or failed the fifth grade. Remember, the identifier **G5** is not a variable; it is a constant value, one value of the enumerated type **Levels**.
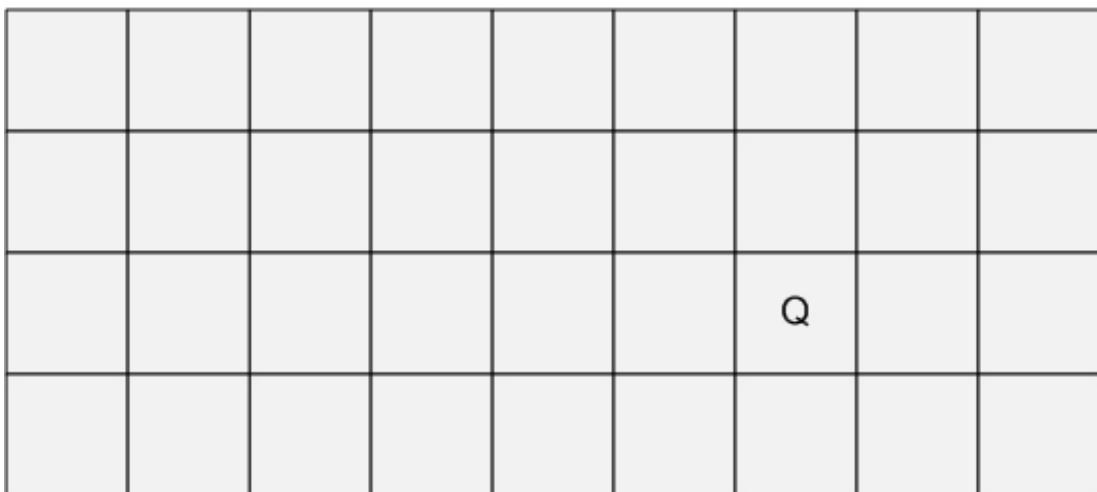
The low limit and high limit of an array's index are called its *bounds*. The bounds of a *static* array in Pascal must be fixed at compile time. The FreePascal compiler must know, when it compiles the program, exactly how large all data items are going to be. FreePascal supports a separate type of array called a dynamic array, which gives you more flexibility when defining arrays. I'll talk about dynamic arrays a little later.

With this in mind, the variable **AreaPerc** deserves a closer look. At first glance, you might think it has a variable for a high bound, but actually, **Districts** is a constant with a value of 14. Writing **[1..Districts]** is no different from writing **[1..14]**, but within the context of the program it assists your understanding of what the array actually represents.
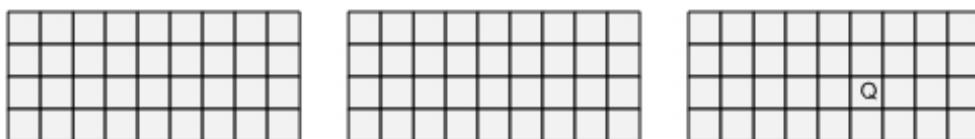
My1DArray : ARRAY[0..8] OF Char;

Element My1DArray[6] contains the "Q"

My2DArray : ARRAY[0..8, 1..4] OF Char;

Element My2DArray[6,3] contains the "Q"

My3DArray : ARRAY[0..8, 1..4, 0..2] OF Char;

Element My3DArray[6,3,2] contains the "Q"

Figure 8.1. Multidimensional Arrays

Multidimensional arrays

Most of the arrays shown in the earlier example are one-dimensional. A one-dimensional array has only one index. A two-dimensional array has two indices. A three-dimensional array has three, and so on. (See Figure 8.1.) An array may have any number of dimensions, but it edges toward bad practice to define an array with more than two or (on the outside) three. Also, the more dimensions an array has, the larger it tends to be and the more memory it uses—and the more likely that parts of it are empty or full of duplicate or rarely-accessed information that does nothing for you but waste memory. There are better, less memory-wasteful ways to handle large, complicated data structures. The old-school solution is linked lists, which I will not cover in this book. Dynamic arrays are another, more modern solution to that problem, and I'll cover those in the next section.

When FreePascal allocates an array in memory, it does not zero the elements of the array, as BASIC would. In general, Pascal does not initialize data items of any type for you. If there was garbage in RAM where the compiler went out to set up an array, the array elements will contain the garbage when the array is allocated. If you wish to zero out or otherwise initialize the elements of an array, you must do it yourself, in your program, before you use the array. This is not difficult, using a **FOR** loop (see Section 9.4 for more on **FOR** loops):

```
FOR I := 1 TO 24 DO Subtotals[I] := 0;
```

Good examples of arrays used effectively can be found in the **ShellSort** procedure in Section 10.3 and the **QuickSort** procedure in Section 10.6.

FreePascal's dynamic arrays

As I mentioned briefly a little earlier, there's a problem with conventional static Pascal arrays: Static arrays are fixed in size, according to their declarations. The compiler must allocate them enough memory to hold all the elements specified in the declaration. That memory is used whether or not you ever actually write data into all of the elements. Perhaps worse, if during a program run your program needs more elements than an array's declaration provides, you're out of luck. What an array gets at compile time is all it will ever have.

FreePascal provides a solution to this problem: *Dynamic arrays* are arrays allocated and maintained in dynamic memory, which is informally called the *heap*. Dynamic arrays were not supported in Turbo Pascal or Borland Pascal, but were added to the language with Delphi and later added to FreePascal. Dynamic memory and pointers (which are how we use dynamic memory) are important but advanced topics that I can't address in this book. Once we move on to Object Oriented Programming (OOP) dynamic memory is crucial, and I'm planning a separate book that takes up both objects and dynamic memory in detail.

A dynamic array is declared without explicit sizes on any of its dimensions. When a program that uses dynamic arrays begins running, the arrays are not only empty, they occupy no space in memory. To use a dynamic array, you must first call a built-in procedure that allocates memory for as many elements as your program requires. For example, declaring a dynamic array of integers would be done this way:

```
VAR
  LuminanceReadings : ARRAY of Integer;
```

FreePascal understands that this is a dynamic array because no bounds are declared for it. Until your code specifies how many elements are in the array, the array occupies no space in memory and cannot be used. You specify the number of elements in the array with a predefined procedure **SetLength**:

```
SetLength(LuminanceReadings, 1000);
```

This procedure call allocates dynamic memory for 1000 integers and associates that block of memory with the variable name **LuminanceReadings**. Once you've allocated memory with **SetLength**, you can use the dynamic array as you would use any array of 1,000 integers. There are some restrictions, the most important of which is this: *The array index begins at 0.* The first element of any dynamic array is always element 0. There's no way for you to set the index of the first element to 1 or any other value.

Dynamic arrays are managed by the runtime code that FreePascal links into your program. When the array goes out of scope (see Chapter 13 for more on scope, which I haven't covered yet) the runtime code automatically deallocates the array's memory on the heap.

There's another, subtler restriction that won't make complete sense until you learn more about dynamic memory and the heap. If you declare the names of two dynamic arrays and allocate one, you can assign the second identifier to the first, like this:

```
VAR
  CrateWeights1, CrateWeights2 = ARRAY of Integer;

SetLength(CrateWeights1, 50);

{ Code here to store values into CrateWeights1 the usual way }

{ Assign the second dynamic array to the first one: }
CreateWeights2 := CrateWeights1;
```

As you would expect, the two arrays now contain exactly the same elements. But there's a very serious catch: *Both identifiers now refer to the same array in dynamic memory.* You only allocated memory for one array, and one is all there is. By assigning a second array to the first, in effect you're giving the allocated array a second name. If you change elements in **CrateWeights2**, those changes will be made to the elements of **CrateWeights1** as well, and vise versa.

This may seem bizarre until you understand the concept of Pascal pointers, which I can't explain in detail in this book. A pointer is basically a memory address. **CrateWeights1** and **CrateWeights2** are in reality two pointers that point to the

same area of memory allocated by the call to **SetLength**.

If you want two identical but independent arrays containing the same elements, you should use the **Copy** function to make an independent copy of **CrateWeights1**:

```
CrateWeights2 := Copy(CrateWeights1);
```

This allocates an entirely new duplicate array on the heap, and once that happens, changes made to **CrateWeights1** will not affect **CrateWeights2** and vise versa.

Likewise, if you call **SetLength** a second time with **CrateWeights2** as the first parameter, you will allocate a second, independent dynamic array somewhere else in memory. This time, the elements of **CrateWeights1** will not be duplicated, and **CrateWeights2** will contain "zero-filled" data until you assign data to it. and making changes to one will have no effect on the other. Note well: It's *not* a good idea to assume that a new dynamic array—or any other new data item—will be automatically zero-filled by the runtime code. If you need to rely on having an array full of zeroes, assign zeroes to all the elements before you begin using it!

Dynamic arrays can be resized by making a call to **SetLength** with a different number of elements. Do not assume that the array will contain the same values after a second call to **SetLength**. Always assume that **SetLength** hands you a brand new array containing undefined values.

You can deallocate a dynamic array at any time by assigning the predefined value **NIL** to the array:

```
LuminanceReadings := NIL;
```

The runtime code will return the space formerly occupied by the array to the heap, so it can be used again for other dynamic variables.

There's one final weirdness about dynamic arrays: In a dynamic array having 100 elements, for example, you should not access element 100. *There is no element 100.* Because dynamic array indexes are always 0-based, the indexes in a 100-element array run from 0-99. If you attempt to read a value at index 100, you will get either a garbage value or a value belonging to some other dynamic variable entirely.

8.5. RECORDS

An array is a data structure composed of a number of identical data items all in

a row and referenced by number. This sort of data structure is handy for dealing with large numbers of the same type of data; for example, values returned from an experiment of some sort. You might have a collection of five hundred temperature readings and need to average them and perform analysis of variance on them. The easiest way to do that is load them into an array and work with the temperature readings as elements of the array.

There is a data structure composed of data items that are not of the same type. It's called a *record,* and it gets its name from its origins as one line of data in a data file.

A record is a structure composed of several data items of different types grouped together. These data items are called the *fields* of the record.

Let's work out a short and much-simplified conceptual example. An auto-repair shop might keep a file on its spare parts inventory. For each part they keep in stock, they need to record its part number, its description, its wholesale cost, retail price, customary stock level and current stock level. All these items are intimately linked to a single physical gadget, (a car part) so to simplify their programming the shop puts the fields together to form a record:

```
TYPE
  PartRec  =  RECORD
                  PartNum, Class  : Integer;
                  PartDescription : String;
                  OurCost         : LongInt;
                  ListPrice       : LongInt;
                  StockLevel      : Integer;
                  OnHand          : Integer;
              END

VAR
  CurrentPart, NextPart  : PartRec;
  PartFile     : FILE OF PartRec;
  CurrentStock : Integer;
  MustOrder    : Boolean;
```

The entire structure becomes a new type with its own name. Data items of the record type can then be assigned, written to files, and otherwise worked with as a single entity without having to explicitly mention all the various fields within the record.

```
CurrentPart := NextPart;  { Assign part record to another }
Read(PartFile,NextPart);  { Read next record from file    }
```

When you need to work with the individual fields within a record, the notation consists of the record identifier followed by a period (".") followed by the field identifier:

```
CurrentStock := CurrentPart.OnHand;
IF CurrentStock < CurrentPart.StockLevel
  THEN MustOrder := True;
```

Accessing individual fields within a record this way is informally called "dotting."

Relational operators may *not* be used on records. To say that one record is "greater than" or "less than" another cannot be defined since there are an infinite number of possible record structures with no well-defined and unambiguous order for them to follow. In the example above we are comparing the fields of two records, not the records themselves. The fields are both integers and can therefore be compared by the "<" operator.

The WITH Statement

As I just explained, fields within a record are accessed by dotting:

```
CurrentPart.OurCost := 1075;      { In pennies! }
CurrentPart.ListPrice := 4185;    { " }
CurrentPart.OnHand := 4;
```

Note the repetition of the record name before the field name. That's logically unnecessary, if we know we're just going to be accessing several fields from the same record in quick succession here. If you have to go down a list of fields within the same record and work with each field, you can avoid specifying the identifier of the record each and every time by using a special statement called the **WITH** statement.

We could simplify assigning values to several fields of the same record by writing the above snippet of code this way:

```
WITH CurrentPart DO
  BEGIN
    OurCost := 1075;
    ListPrice := 4185;
    OnHand := 4;
  END;
```

The space between the **BEGIN** and **END** is the "scope" of the **WITH** statement. Within that scope, the record identifier need not be given to work with the fields

of the record named in the **WITH** statement. (If you are very new to Pascal, you might return to this section after reading the general discussion on statements in Section 9.1. **WITH** statements are subject to the same rules that all types of Pascal statements obey.)

WITH statements need not have a **BEGIN/END** unless they contain more than a single statement. A **WITH** statement may include only a single statement to work with a record if that single statement contains several references to fields within a single record:

```
WITH CurrentPart DO VerifyCost(OurCost,ListPrice,Check);
```

In this example, **VerifyCost** is a procedure that takes as input two price figures and returns a value in **Check**. Without the **WITH** statement, calling **VerifyCost** would have to be done this way:

```
VerifyCost(CurrentPart.OurCost,CurrentPart.ListPrice,Check);
```

The **WITH** statement makes the statement crisper and much easier to understand. By using **WITH**, we can hide the unnecessary details of what larger record we're working with during the time that we're working on a small-scale with the fields of that record. There's no need to "see" the name of the embracing record every time we access a field, so the smart thing to do is put the name of the record off to one side so it doesn't get in the way. That's the spirit in which **WITH** was created.

Nested Records

A record is a group of data items taken together as a named data structure. A record is itself a data item, and so records may themselves be fields of larger records. Suppose the repair shop we've been speaking of expands its parts inventory so much that finding a part bin by memory gets to be difficult. There are ten aisles with letters from A through J, with the bins in each aisle numbered from 1 up. To specify a location for a part requires an aisle character and a bin number. The best way to do it is by defining a new record type:

```
TYPE
  PartLocation = RECORD
                   Aisle : 'A'..'J';
                   Bin   : Integer
                 END;
```

Since each part has a location, type **PartRec** needs a new field:

```
TYPE
  PartRec  = RECORD
               PartNum, Class  : Integer;
```

```
        PartDescription : String;
        OurCost         : LongInt;
        ListPrice       : LongInt;
        StockLevel      : Integer;
        OnHand          : Integer;
        Location        : PartLocation { A record! }
    END;
```

Location is "nested" within the larger record. To access the fields of **Location** you need two periods:

```
LookAisle := CurrentPart.Location.Aisle
```

If the outermost record is specified by a **WITH** statement you might have an equivalent statement like this:

```
WITH CurrentPart DO LookAisle := Location.Aisle;
```

WITH statements are fully capable of handling many levels of record nesting. You may, first of all, nest **WITH** statements one within another. The following compound statement is equivalent to the previous statement:

```
WITH CurrentPart DO
  WITH Location DO LookAisle := Aisle;
```

The **WITH** statement also allows a slightly terser form to express the same thing:

```
WITH CurrentPart, Location DO LookAisle := Aisle;
```

For this syntax you must place the record identifiers after the **WITH** reserved word, separated by commas, *in nesting order*. That is, the name of the outermost record is on the left, and the names of records nested within it are placed to its right, with the "innermost" nested record placed last.

Records were originally essential in their use as "slices" of a disk file. This is much less useful than it once was, with simple databases like SQLite shipped free with FreePascal and Lazarus. I won't be discussing file I/O in this book for that reason: It's been largely subsumed by database technology.

Records lost even more of their usefulness when objects were added to Pascal with Turbo Pascal 5.5. Objects resemble records, but may contain procedures and functions as well as data. (Objects have other, subtler tricks too, and I really can't cover them in this introductory book.)

One final note on records: from its original definition Pascal has supported "variant records," a feature that allows a record to have two or more different

structures depending on the value of a field in the record. I've dropped the description of variant records from this book because almost no one uses them anymore. An advanced Pascal feature called *variants* broadens the concept beyond records, as does a feature of object-oriented programming called *polymorphism*. Both of these are advanced topics that I hope to cover in a brand new book at some point.

If you find you need a solid explanation of variant records, copies of my 1993 print book *Borland Pascal 7 from Square One* treats them at length and can be had on the used book markets.

8.6. STRINGS

Manipulating words and lines of text is a fundamental function of a computer program. At minimum, a program must display messages like "Press RETURN to continue:" and "Processing completed." In Pascal, as in most computer languages, a line of characters to be taken together as a single entity is called a *string*.

Standard Pascal PAOC strings (fixed length)

The original ISO Standard Pascal had very little power to manipulate strings. In Standard Pascal, there is nothing formally referred to as type **STRING**, as in Turbo Pascal and FreePascal. To hold text strings, Standard Pascal uses a packed array of characters, sometimes abbreviated as PAOC:

```
TYPE
  PAOC25 = PACKED ARRAY[1..25] OF Char;
```

This is a typical definition of a PAOC type. Of course, it doesn't have to be 25 characters in size; within the constraints of available memory it can be as large as you like. The reserved word **PACKED** is a holdover from ancient mainframe computers; on 8 and 16-bit personal computers it served no purpose. In mainframe computers with 32 and 64-bit words, the word **PACKED** instructs the compiler to store as many characters as will fit in one machine word, rather than using one machine word per character. Using **PACKED** on a mainframe could reduce the size of a string by a factor of four to eight times. Modern Pascal compilers always store a Char value in one byte no matter what the word size of the computer it runs on. The word **PACKED** is still necessary, however, to define a Standard Pascal string.

What can be done with a PAOC-type string? Not much. A string constant can be assigned to it, if the constant is exactly the same size as the definition of the PAOC:

```
VAR
  ErrorMsg : PAOC25;
```

```
ErrorMsg := 'Warning! Bracket missing!';   { This is OK }
ErrorMsg := 'Warning! Comma missing!';     { Illegal!  }
```

The second string constant is two characters short of 25 long, so the compiler will display an error message.

The second string constant could not be considered a **PAOC25** because it is only 23 characters long; hence the type mismatch error. Of course, you could have padded out the second constant with spaces, and the padded constant would have been acceptable.

You can compare two PAOC-type strings with the relational operators, read and write them from files, and print them to the screen. And that's where it ends. All other manipulations have to be done on a character-by-character basis, as though the string were just another array of any simple type.

Note well: PAOC-type strings are now decades obsolete. The only time to use them is in situations where you are forced to deal with code in extremely old programs and have to add features or fix bugs.

Short strings

All modern Pascal implementors, including those who wrote Turbo Pascal and FreePascal, have implemented character strings as what are called *variable-length strings*. Like PAOC-type strings, variable-length strings are arrays of characters, but they are treated by the compiler in a special way. There are several distinct varieties of variable-length strings in FreePascal, including some that cater to Unicode *code points*, which are characters that require two (or sometimes four) bytes to express them.

The original variable-length strings defined by Turbo Pascal are limited to 255 characters, and are called *short strings*. Short strings have a *logical length* that varies depending on what you put into the string. Strings of different logical lengths may be assigned to one another as long as the real, physical lengths of the strings are not exceeded.

A short string has two lengths: a physical length and a logical length. The physical length is the amount of static memory that the string actually occupies. This length is set at compile time and never changes. The logical length is the number of characters currently considered to be stored in the string. This can change as you work with the string. The logical length (which from now on we will simply call the length) is a numeric value stored as part of the string itself and can be read by your code.

A string variable is defined using the reserved word **STRING**. The default physical length for a string defined as **STRING** in FreePascal is 255 characters.

(This differs from Turbo Pascal V3.0 and earlier, in which the **STRING** type had no default length, and derived string types of some given size had to be explicitly declared in the **TYPE** section of your programs.) You may not always need a string that physically large (for example, a telephone number fits comfortably in less than 20 characters) and you may define smaller string types to save memory by placing the physical size after the reserved word **STRING** in brackets:

```
VAR
  Message : String[80];   { Physical size = 80 }
  Name    : String[30];   { Physical size = 30 }
  Address : String[30];   { Physical size = 30 }
  State   : String[2];    { Physical size = 2  }
```

The legal range of physical lengths for variable-length strings is 1 to 255 characters.

You may also define separate string types as strings of different physical lengths. This is a much better way to deal with strings shorter than 255 characters:

```
TYPE
  String80 = String[80];
  String30 = String[30];
  Buffer   = String[255];
```

Once you have defined these types, declare all string variables that are to have a physical length of 30 characters as type **String30**. This way, all such strings will have identical types and not simply compatible types.

So what is a short string, physically? A string is an array of characters indexed from 0 to the physical length. Character 0 is special, however: It is the *length byte* and it holds the logical length of the string at any given time. The length byte is set by the runtime code when you perform an operation on a string that changes its logical length. Assignment and string concatenation using the "+" operator are two ways to do this. The **Concat** function is another that I'll discuss later.

```
MyString := '';                           { MyString[0] = 0  }
MyString := 'Frodo';                      { MyString[0] = 5  }
MyString := 'Alfred E. ' + 'Newmann';     { MyString[0] = 17 }
```

Strings may be accessed as though they were in fact arrays of characters. You can reference any character in the string, including the length byte, with a normal array reference:

```
VAR
  MyString  : String[15];
```

```
  CharS     : Integer;
  OUTChar   : Char;

MyString := 'Galadriel';
CharS     := ORD(MyString[0]);  { CharS now equals 9    }
OUTChar   := MyString[6];       { OUTChar now holds 'r' }
```

Even though the runtime code treats the length byte as a number, it is still an element in an array of **Char** and thus cannot be assigned directly to type **Byte** or **Integer**. To assign the length byte to a numeric variable, you must use the **ORD** transfer function, which is Pascal's orderly way of transferring a character value to a numeric value. (See Section 11.5.)

Having told you that this is possible, let me warn you: *Don't do it*. It was common enough in the Turbo Pascal era, because short strings were more or less all there were in the string department. String handling in modern Pascal compilers has gradually moved away from short strings toward (much) longer strings allocated dynamically, as dynamic arrays are. I'll explain dynamic strings in the next section. The warning is that dynamic strings don't have a length counter at element 0, so accessing element 0 won't accomplish anything useful, and (depending on how the compiler is configured) may generate an error.

No. Use the predefined string-handling functions and procedures built into FreePascal instead. The **Length** function is a good example, and makes directly accessing a short string's length counter unnecessary. It returns the current logical length of a string variable of any type:

```
CharCount := Length(MyString);   { CharCount now equals 9 }
```

I'll discuss **Length, Concat,** and all the other built-in string handling functions and procedures in detail in Chapter 12.

Characters and short strings are compatible in some limited ways. You can assign a character value (stored either in constant form or as a **Char** variable) to a string variable. The string variable then has a logical length of one:

```
VAR
  OUTChar : Char;

MyString := 'A'            { Logical length = 1 }
MyString := OUTChar;       { Ditto }
```

The reverese, however, isn't true: A string (even one having a length of zero or one) cannot be assigned to a variable of type **Char**.

You can compare a string variable to a character literal:

```
IF MyString = 'A' THEN StartProcess;
```

In any string comparison, the two strings must be identical in both length and content for a **TRUE** value to be generated by the expression.

It's possible to assign a string to another string with a shorter physical length. This will cause neither a compile time nor a run time error. What it will do is truncate the data from the larger string to the maximum physical length of the smaller string.

You can concatenate a character literal or variable to a string variable, using either the **Concat** function or (more commonly) the "+" operator. **Concat** (which I'll describe briefly in section 12.2) is a holdover from ancient times, and I recommend using the "+" operator instead. Be aware of it if you have to deal with older Pascal code. For new code, use "+".

ANSI Strings

Short strings were first developed for UCSD Pascal in 1978, and have been supported by nearly all Pascal compilers since then. As I explained earlier, short strings may be at most only 255 characters long. That's useful, but still limiting. FreePascal's **ANSIString** type (introduced with Delphi 2.0) eliminates any practical length limitations. **ANSIString** variables can be as long as 4,294,967,295 characters, which is plenty, and in fact represent more memory than a lot of low-end computers actually provide. You can load entire text documents into a single **ANSIString** variable for word counting or other document-wide operations. Four gigabytes is a *lot* of room; this entire book will fit in a single **ANSIString** variable fifty times over.

Internally, however, the two string types are vastly different. An **ANSIString** variable is a slightly enhanced dynamic array of characters. It's stored in dynamic memory on the heap, just as dynamic arrays are. An **ANSIString** variable is really a pointer to the string data on the heap. Ahead of the string data is a 32-bit length counter, plus a reference counter. (More on this at the end of this section.) Also, every **ANSIString** is guaranteed to be terminated by a null character. Any string operation that changes the number of characters in the variable will move the null appropriately. Null termination is there primarily to make **ANSIString** variables compatible with **PChar** strings (see below) which are generally used when Pascal code needs to interface with code from other languages that use **PChar** as their primary string type, like C and C++.

You can use all of the standard string functions with **ANSIString**, including **Length**. In fact, if you stick to standard functions (rather than attempting to inspect and manipulate the data on the heap by other means) and respect the 255-character limitation on short strings, there is no significant difference in how the two string

types are used.

Short string or ANSIString?

At this writing (2016) short strings are used only rarely in new code. By default, when you declare a variable as type **STRING**, FreePascal will treat the variable as an **ANSIString**. In order to force the compiler to treat a **STRING** as type **ShortString**, you need to use the $H compiler directive. By default, the state of $H is plus, that is, $H+. When $H is plus, the compiler treats **STRING** as **ANSIString**. When $H is minus, the compiler treats **STRING** as **ShortString**. You can change the state of $H at will in the **VAR** section of a program:

```
VAR
  GreatBigString   : STRING; {Will compile as ANSIString }
{$H-}
  AncientString    : STRING; {Will compile as ShortString }
{$H+}
  AnotherBigString : STRING; {Will compile as ANSIString }
```

There's one exception to this rule: When you include a length value in the string variable definition, FreePascal will treat the variable as a short string, irrespective of the current state of the $H directive. The string definition below will always compile **StreetName** as **ShortString**:

```
VAR
  StreetName : STRING[80]; {Will always compile as ShortString }
```

Of course, if you turn $H to minus, make sure you turn it back to plus, or all **STRING** definitions from then on will be compiled as **ShortString**.

ANSIString references

An **ANSIString** variable is a dynamic variable, existing entirely on the heap. The variable is in fact a pointer to the string's data, preceded by two 32-bit values: a length counter, and a reference counter. The reference counter is part of FreePascal's machinery supporting something called *lifetime management*. Lifetime management applies to several data types that reside on the heap, especially dynamic string types (there are several) and dynamic arrays. The idea is that data blocks on the heap that are no longer being pointed to (referenced) by a pointer of some sort should not be allowed to take up memory forever. If you've never dealt with Pascal pointers before, this section may be difficult to follow. I'll deal with them at length in a future book; do the best you can here, or come back after learning more about the heap.

When a program is executed, the runtime code sets up variables in memory. A dynamic variable of type **ANSIString** begins life as a null pointer. (A null pointer is a pointer with all bits set to 0.) No heap memory is initially involved. Only when some value is assigned to an **ANSIString** variable is memory allocated on the heap to hold the value. At that point, the variable's reference counter on the heap is set to 1. If a second **ANSIString** variable is assigned the value of the first **ANSIString**, a second memory block is *not* created. Instead, the second **ANSIString** is pointed to the same memory block owned by the first. You do not have two copies of the string data. You have two pointers pointing to the *same* copy of the string data. With two pointers pointing at the same data, the reference counter preceding the data on the heap is given the value 2. See how this works in he code below:

```
VAR
  GreenString, RedString : ANSIString;
  . . .

  { GreenString and RedString begin life as null pointers. }
GreenString := 'Green';
  { GreenString is now allocated on the heap. Ref counter = 1 }

  RedString := GreenString;
  { RedString now points to GreenString's data. Ref counter = 2 }

  RedString := 'Red';
  { RedString now has its own heap data block. GreenString's   }
  { ref counter is now back to 1. RedString's ref counter is 1. }

  GreenString := '';
  { GreenString is now empty. Ref count is 0. The "garbage      }
  { collection" code releases GreenString's data block. Now     }
  { GreenString has no heap data and is again a null pointer. }
```

The key point is that when an **ANSIString** becomes empty, the data block it once had on the heap becomes useless, and its memory wasted. Lifetime management uses its "garbage collector" function to de-allocate that memory so that the memory may be used by other dynamic variables in the program.

If this sounds scary, relax. It's all handled automatically by the runtime code, and (especially while you're still a beginner) you don't need to be aware of it. Once you learn about Pascal pointers, I guarantee that it will all make a great deal more sense!

C-style PChar null-terminated string support

To support easier programming of Microsoft Windows applications in Pascal back in the 1990s, Borland added a unit to the product that supports C-style null-terminated

strings. The **Strings** unit contains a number of short procedures that manipulate C-style strings, and if you wish to use C-style strings you must include the **Strings** unit in your **USES** statement. (More about units and **USES** in Chapter 15.) FreePascal includes the **Strings** unit. The C-style string type is **PChar**. It's basically a pointer to a sequence of characters set out in static memory at compile time. **PChar** strings are not stored in dynamic memory on the heap, as are **ANSIString** strings. Using various string functions and operators in the **Strings** unit, characters are stored in the area of memory allocated to the **PChar**, and after the last character containing real data (not after the last byte allocated for the string!) there is a null character, which is a binary 0. The null character moves up and down the string as characters are written to the string. There is no length counter and no reference counter. To determine the length of a **PChar** string at any given time, you must literally scan the string, counting characters until you hit the null. The Strings unit has functions that perform this and many other services.

Using C-style strings requires that you be proficient in the use of pointers, which is a great deal to ask of beginning Pascal programmers, especially when the use case for C-style strings is so narrow. I recommend not bothering with them unless you are forced to deal with Microsoft Windows API calls or other code libraries that were designed to be called from the C language. You can find more about them online, and I will not be discussing them further in this book.

"Wide" strings

One other category of string types needs to be mentioned, even though I'm not going to cover them in detail in this book. (I may add a more detailed description in a future update.) Those are the "wide" strings, which are strings of Unicode characters.

Unicode is an industry standard laying out a character format for characters beyond the 255 characters present in the ASCII character set used in English and Western European software. Each Unicode character is defined in two bytes rather than one. This allows many other alphabets (such as Cyrillic or Arabic) to be used in software.

Type **UnicodeString** is much like **ANSIString**, except that two adjacent bytes are used to represent each character, and the string as a whole is terminated with two null characters.

There is a great deal more to Unicode than this, and for the time being I must refer you online (for the big picture) and to the FreePascal doc for details.

8.7. Typed constants

Standard Pascal only allows simple constants: Integers, characters, reals, Booleans, and strings. Turbo Pascal introduced *typed constants*, meaning constants with an explicit type that are initialized to some specific value. FreePascal continues and expands that support.

Calling typed constants "constants" is not entirely fair. Real constants are hardcoded "in-line" into the machine code produced by the compiler, with an actual physical copy of the constant dropped in everywhere it is named. The constant thus exists at no single address. Turbo Pascal's typed constants are actually static variables that are initialized at runtime to values taken from the source code. They exist at one single address, which is referenced anytime the typed constant is used.

Typed constants also violate the most fundamental proscription of constants in all languages: They may be changed during the course of a program run. Of course, you are not obligated to alter typed constants at runtime, but the compiler will not stop you if you try.

With that in mind, it might be better to think of typed constants as a means of forcing the compiler to initialize complicated data structures. Standard Pascal has *no* means of initializing variables automatically. If values are to be placed into variables, *you* must place them there somehow, either from assignment statements or by reading values in from a file. For example, you could initialize an array of fifteen integers this way:

```
VAR
  Weights : ARRAY[1..15] OF Integer;


Weights[1]  := 17;
Weights[2]  := 5;
Weights[3]  := 91;
Weights[4]  := 111;
Weights[5]  := 0;
Weights[6]  := 44;
Weights[7]  := 16;
Weights[8]  := 3;
Weights[9]  := 472;
Weights[10] := 66;
Weights[11] := 14;
Weights[12] := 38;
Weights[13] := 57;
Weights[14] := 8;
Weights[15] := 10;
```

For fifteen values this may seem manageable. But suppose you had fifty values?

Or a hundred? At that point typed constants become very attractive. This same array could be initialized as a structured constant like so:

```
CONST
  Weights : ARRAY[1..15] OF Integer =
    (17,5,91,111,0,44,16,3,472,66,14,38,57,8,10);
```

The form of a typed constant definition is this:

\<identifier\> : \<type\> = \<values\>

Constant definitions, by convention, represent the first section of a Pascal program, coming before type and variable definitions. Because FreePascal allows multiple **CONST** keywords within a single program, you may have a separate **CONST** declaration section for typed constants *after* the type declaration section. This allows you to declare your own custom type definitions first, and then create constants having your custom types.

Array constants

The numeric array example above is a simple, one-dimensional array constant. Its values are placed, in order, between parentheses, with commas separating the values. *You must give one value for each element of the array constant.* The compiler will not allow you to initialize some values of an array and leave the rest "blank." You must do all of them or none at all. If the number of values you give does not match the number of elements in the array, the compiler will display an error message.

If you only need to initialize a few values out of a large array, (leaving the others undefined) it makes more sense to go back to individual assignment statements.

You may also define *multidimensional* array constants. The trick here is to enclose each dimension in parentheses, with commas separating both the dimensions and the items. A single pair of parentheses must enclose the entire constant. The innermost nesting level represents the rightmost dimension from the array declaration. An example will help:

```
CONST
  Grid : ARRAY[0..4,0..3] OF Integer =
             ((4,6,2,1),
              (3,9,8,3),
              (1,7,7,5),
              (4,1,7,7),
              (3,1,3,1));
```

This is a two-dimensional array of integer constants, arranged as five rows of

four columns, and might represent game pieces on a game grid. Adding a third dimension to the game (and the grid) would be done this way:

```
CONST
  Space : ARRAY[0..7,0..4,0..3] OF Integer =

(((4,6,2,1),(3,9,8,3),(1,7,7,5),(4,1,7,7),(3,1,3,1)),
 ((1,1,1,1),(1,1,1,1),(1,1,1,1),(1,1,1,1),(1,1,1,1)),
 ((2,2,2,2),(2,2,2,2),(2,2,2,2),(2,2,2,2),(2,2,2,2)),
 ((3,3,3,3),(3,3,3,3),(3,3,3,3),(3,3,3,3),(3,3,3,3)),
 ((4,4,4,4),(4,4,4,4),(4,4,4,4),(4,4,4,4),(4,4,4,4)),
 ((5,5,5,5),(5,5,5,5),(5,5,5,5),(5,5,5,5),(5,5,5,5)),
 ((6,6,6,6),(6,6,6,6),(6,6,6,6),(6,6,6,6),(6,6,6,6)),
 ((7,7,7,7),(7,7,7,7),(7,7,7,7),(7,7,7,7),(7,7,7,7)));
```

The values given for the two-dimensional array have been retained here to see how the array has been extended by one dimension. Note that the list of values in an array constant must begin with the same number of left parentheses as the array has dimensions. Remember, also, that every element in the array must have a value in the array constant declaration.

Notice that this mechanism allows you to initialize 160 different integer values in a relatively small space. Imagine what it would have taken to initialize this array with a separate assignment statements for each array element!

Record constants

Record constants are handled a little bit differently. You must first declare a record type and then a constant containing values for each field in the record. The list of values must include the name of each field, followed by a colon, and then the value for that field. Items in the list are separated by semicolons. As an example, consider a record containing configuration values for a serial terminal program:

```
TYPE
  BPS         = (B110,B300,B1200,B2400,B4800,B9600);
  ParityType  = (EvenParity,OddParity,NoParity);
  TermCFG     = RECORD
                  LocalAreaCode : String[3];
                  UseTouchtones : Boolean;
                  DialOneFirst  : Boolean;
                  BaudRate      : BPS;
                  BitsPerChar   : Integer;
                  Parity        : ParityType
                END;
```

```
CONST
  Config : TermCFG =
                (LocalAreaCode : '716';
                 UseTouchtones : True;
                 DialOneFirst  : True;
                 BaudRate      : B1200;
                 BitsPerChar   : 7;
                 Parity        : EVEN_PARITY);
```

The structured constant declaration for **Config** must come after the type definition for **TermCFG**, otherwise the compiler would not know what a **TermCFG** was. Note that FreePascal allows multiple **CONST** sections, and will allow you to place a **CONST** section after a **TYPE** section. This would not be allowed under Standard Pascal. Also, note that there is no **BEGIN/END** bracketing in the declaration of Config. The parentheses serve to set off the list of field values from the rest of your source code.

Set constants

Declaring a set constant is not very different from assigning a set value to a set variable:

```
CONST
 Uppercase : SET OF Char = ['A'..'Z'];
```

The major difference is the notation used to represent non-printable characters in a **SET OF Char**. Characters that do not have a printable symbol associated with them may ordinarily be represented in a set builder by the **Chr** transfer function:

```
MySet := [Chr(7),Chr(10),Chr(13)];
```

Important: The **Chr** notation shown above will not work when declaring set constants. You have two alternatives:

1) Express the character as a control character by placing a caret symbol (^) in front of the appropriate character. The bell character, **Chr(7)**, would be expressed as ^G.

2) Express the character as its ordinal value preceded by a pound sign (#). The bell character would be expressed as #7. This notation is more useful for expressing characters falling in the "high" 128 bytes of type **Char**, corresponding to the line-drawing, mathematical, and foreign language characters on PCs.

For example, the set of whitespace characters is a useful set constant:

```
CONST
```

```
  Whitespace : SET OF Char = [#8,#10,#12,#13,' '];
```

or, alternatively:

```
CONST
  Whitespace : SET OF Char = [^H,^J,^L,^M,' '];
```

The following three routines show you how to use set constants in simple character-manipulation tools. If you do a lot of character manipulation and find a great deal of use for these three set constants, you might also declare them at the global program level so that any part of the program can use them. Remember that, declared as it is here locally to the individual functions, the **Whitespace** constant cannot be accessed from outside the function! This is a consequence of the scoping rules of procedures and functions, which I'll cover in detail in Chapter 13.

```
FUNCTION CapsLock(Ch : Char) : Char;

CONST
  Lowercase : SET OF Char = ['a'..'z'];

BEGIN
  IF Ch IN Lowercase THEN CapsLock := Chr(Ord(Ch)-32)
    ELSE CapsLock := Ch
END;



FUNCTION DownCase(Ch : Char) : Char;

CONST
  Uppercase : SET OF Char = ['A'..'Z'];

BEGIN
  IF Ch IN Uppercase THEN DownCase := Chr(Ord(Ch)+32)
    ELSE DownCase := Ch
END;



FUNCTION IsWhite(Ch : Char) : Boolean;

CONST
  Whitespace : SET OF Char = [#8,#10,#12,#13,' '];

BEGIN
  IsWhite := Ch IN WhiteSpace
END;
```

CHAPTER 9.
STRUCTURING CODE

Controlling the flow of program logic is one of the most important facets of any programming language. Conditional statements that change the direction of the flow of control, looping statements that repeat some action a number of times, switch statements that pick one course out of many based on some controlling value, all make useful programs possible.

Pascal, furthermore, would like you the programmer to direct the flow of control in a structured, rational manner, so the programs you write are easy to read and easy to change when they need changing. For this reason, wild-eyed zipping around inside a program is difficult in Pascal.

The language syntax itself suggests with some force that programs begin at the top of the page and progress generally downward, exiting at the bottom when the work is done. Multiple entry and exit points and unconditional branching via **GOTO** are more trouble to set up—which is just as well, because they can be a lot more trouble to understand and debug when they don't go quite where you want them to, when you want them to, and therefore do what you want them to.

And making things go where you want them to is the fundamental purpose of this admittedly large chapter.

9.1. BEGIN, END, and Compound Statements

We have already looked at several simple types of statements like assignment statements and data definition statements. In Pascal you frequently need to group a number of statements together and treat them as though they were a single statement. The means to do this is the pair of reserved words **BEGIN** and **END**. A group of statements between a **BEGIN** and **END** pair becomes a *compound statement*. The bodies of procedures and functions, and of programs themselves, are compound statements:

```
PROGRAM Rooter;

VAR
  R,S : Real;

BEGIN
  Writeln('>>Square root calculator<<');
  Writeln;
  Write('>>Enter the number: ');
  Readln(R);
  S := Sqrt(R);
  Writeln('  The square root of ',R:7:7,' is ',S:7:7,'.')
END.
```

The statements bracketed by **BEGIN** and **END** in the above example are all simple statements, but that need not be the case. Compound statements may also be parts of larger compound statements:

```
PROGRAM BetterRooter;

VAR
  R,S : Real;

BEGIN
  Writeln('>>Better square root calculator<<');
  Writeln;
  R:=1;
  WHILE R<>0 DO
    BEGIN
      Writeln('>>Enter the number (0 to exit): ');
      Readln(R);
      IF R<>0 THEN
      BEGIN
        S := Sqrt(R);
        Writeln('  The square root of ',R:7:7,' is ',S:7:7,'.');
        Writeln
      END
    END;
  Writeln('>>Square root calculator signing off...')
END.
```

This program contains one compound statement nested inside another, and both nested within a third compound statement that is the body of the program itself.

This is a good place to point out the "prettyprinting" convention that is virtually always used when writing Pascal code. The rule on prettyprinting turns on compound statements: Each compound statement is indented two spaces to the right of the rest of the statement in which it is nested.

It's also crucial to remember that *prettyprinting is ignored by the compiler.* It is strictly a typographical convention to help you sort out nested compound statements by "eyeballing" rather than counting **BEGIN**s and **END**s. You could as well (as some do) indent by three or more spaces instead of two. You could also (as some do) not indent at all. The compiler doesn't care. But the readability of your program will suffer if you don't use prettyprinting.

There is one other thing about compound statements that might seem obvious to some but very un-obvious to others: Compound statements can be used anywhere a simple statement can. Anything between a **BEGIN**/**END** pair is treated syntactically by the compiler just as single simple statement would be.

Compound statements may also be bounded by the reserved words **REPEAT** and **UNTIL**, as I'll show a little later in this chapter.

9.2. IF/THEN/ELSE

A *conditional statement* is one that directs program flow in one of two directions based on a Boolean value. In Pascal the conditional statement is the **IF/THEN/ELSE** statement. The **ELSE** clause is optional, but every **IF** reserved word must have a **THEN** reserved word associated with it. In its simplest form, such a statement is constructed this way:

```
IF <Boolean expression> THEN <statement>
```

The way this statement works is almost self-explanatory from the logic of the English language: If the Boolean expression evaluates to True, then <statement> is executed. If the Boolean expression evaluates to False, control "falls through" to the next statement in the program.

Adding an **ELSE** clause makes the statement look like this:

```
IF <Boolean expression> THEN <statement1>
  ELSE <statement2>
```

Here, if the expression evaluates to True then <statement 1> is executed. If the expression evaluates to False, then <statement 2> is executed. If an **ELSE** clause exists, you can be sure that one or the other of the two statements will be executed.

Either or both of the statements associated with **IF/THEN/ELSE** may be compound statements. Remember that a compound statement may be used anywhere a simple statement may. For example:

```
IF I < 0 THEN
  BEGIN
    Negative := True;        { Set negative flag      }
    I := Abs(I)              { Take abs. value of I   }
  END                        { Never semicolon here!  }
ELSE Negative := False;      { Clear negative flag    }
```

An important point to remember: There is **no** semicolon after the **BEGIN/END** compound statement. The entire code fragment above is considered a single IF statement. A crucial corollary: There is *never* a semicolon immediately before the **ELSE** reserved word in an **IF/THEN/ELSE** statement. Adding one will give you a "freestanding **ELSE**," which is meaningless in Pascal and will trigger a syntax error. The only place you'll ever find a semicolon immediately before an **ELSE** word is inside a **CASE** statement, as I'll explain a little later in connection with **CASE** statements.

Nested IF Statements

Since an IF statement is itself a perfectly valid statement, it may be one or both of the statements contained in an **IF/THEN/ELSE statement**. **IF**s may be nested as deeply as you like—but remember that if someone reading your code must dive too deeply after the bottommost **IF**, he or she may lose track of things and drown before coming up again. If the sole purpose of multiply-nested **IF**s is to choose one alternative of many, it is far better to use the **CASE** statement, which will be covered in Section 9.3. Structurally, such a construction looks like this:

```
IF <Boolean expression1> THEN
  IF <Boolean expression2> THEN
    IF <Boolean expression3> THEN
      IF <Boolean expression4> THEN
        IF <Boolean expression5> THEN
          <statement>;
```

The bottom line here is that all Boolean expressions must evaluate to True before <statement> is executed.

Such a downward escalator of IFs is often hard to follow. Sharp readers may already be objecting that this same result could be done with **AND** operators:

```
IF <Boolean1> AND <Boolean2> AND <Boolean3>
  AND <Boolean4> AND <Boolean5> THEN
  <statement>;
```

This is entirely equivalent to the earlier nested **IF** with one sneaky catch: Here, *all* the Boolean expressions are evaluated before a decision is reached on whether or

not to execute <statement>. With the nested **IF**, the compiler will stop testing as soon as it encounters a Boolean expression that turns up False. In other words, in a nested **IF**, if <Boolean3> is found to be False, the compiler never even evaluates <Boolean4> or <Boolean5>.

Nitpicking? No! There are times when in fact the reason for <Boolean3> might be to make sure <Boolean4> is not tested in certain cases. Divide by zero is one of those cases. Consider this:

```
IF AllOK THEN
  IF R > PI THEN
    IF S <> 1 THEN
      IF (R / ((S*S)-1) < PI) THEN
        CalculateRightAscension;
```

Here, a value of **S** = 1 will cause a divide-by-zero error if the code attempts to evaluate the next expression. So the code *must* stop testing if **S** <> 1 turns up False or risk crashing the program with a runtime divide-by-zero error.

With nested **IF**s you can determine the sequential order in which a series of tests is done. A string of **AND** operators between Boolean expressions may evaluate those expressions in any order dictated by the code generator's optimization logic. If one of your tests carries the hazard of a run-time error, use nested **IF**s.

Nested ELSE/IFs

The previous discussion of nested IFs did not include any **ELSE** clauses. Nesting **IF**'s does not preclude **ELSE**s, though the use and meaning of the statement changes radically. Our previous example executed a series of tests to determine whether or not a single statement was to be executed. By using nested **ELSE/IF**s you can determine which of many statements is to be executed:

```
IF <Boolean1> THEN <statement1>
  ELSE
    IF <Boolean2> THEN <statement2>
      ELSE
        IF <Boolean3> THEN <statement3>
          ELSE
            IF <Boolean4> then <statement4>
              ELSE <statement5>;
```

The code will descend the escalator, and as soon as it finds a Boolean with a value of True, it will execute the statement associated with that Boolean. The tests are performed in order, and even if <Boolean4> is True, it will not be executed (or <Boolean4> even evaluated) if <Boolean2> is found to be True first.

The final **ELSE** clause is not necessary; it provides a "none of the above" choice, in case none of the preceding Boolean expressions turned out to be true. You could simply omit it and control would fall through to the next statement without executing any of the statements contained in the larger **IF** statement.

As with nested **IF**s described above, nested **ELSE/IF**s allow you to set the order of the tests performed, so that if one of them carries the danger of a run-time error, you can defuse the danger with an earlier test for the dangerous values.

The **CASE** statement is a shorthand form of nested **ELSE/IF**s in which all of the Boolean expressions are of this form: <value> = <value> and the type of all <value>s is identical. We'll look at the **CASE** statement in detail in Section 9.3.

Short-circuit Boolean evaluation

Apart from nesting IF statements, there is another, considerably less portable means of dictating the order in which the compiler evaluates Boolean expressions. It involves a compiler optimization technique introduced with Turbo Pascal 4.0 called "short circuit Boolean evaluation."

A few paragraphs back we looked at a nested **IF** construction that avoided the possibility of a divide-by-zero error by testing for a divide-by-zero condition before performing the actual divide operation. Consider that nested **IF** expressed as a single Boolean expression:

```
IF AllOK    AND
   (R > PI) AND
   (S <> 1) AND
   (R / ((S*S)-1) < PI)        { Could trigger divide-by-zero! }
THEN CalculateRightAscension;
```

In most cases, the compiler will evaluate all four of the Boolean subexpressions before combining them into a single Boolean value as the result of the entire, larger expression. As we mentioned earlier, if **S** ever takes on a value of 1, the fourth subexpression will trigger a divide-by-zero error, since when **S** equals 1, **R** is divided by **((S*S)-1)** which equals zero.

By turning on short-circuit Boolean evaluation, the compiler is forced to evaluate Boolean expressions from left to right, and it will stop evaluation as soon as it is sure that testing further will not change the ultimate value of the expression.

How can it be sure that further testing won't change things? Think about the meaning of the **AND** operator. If any number of Boolean subexpressions are **AND**ed together, *a single False value will force the whole expression to False*. So once that first False turns up in evaluating an expression from left to right, the whole

expression will be False no matter what else waits to be evaluated further to the right.

This technique is called "short circuit," because it quits when possible before evaluating an entire expression. The primary reason for it is that it can make your programs run more quickly if there are many Boolean expressions to be evaluated. Any time you can perform the same job by executing less code, the job will go more quickly.

However, the side benefit of allowing you to arrange your Boolean expressions so that "dangerous" subexpressions are to the right of "sentinel" subexpressions that guard against the dangerous condition is perhaps more generally useful. Returning to the example above: If short-circuit Boolean evaluation is enabled, the compiler will evaluate the subexpression **(S <> 1)**. If **S** is equal to 1, evaluation stops there, and the dangerous expression **(R / ((S*S)-1) < PI)** will never be evaluated, eliminating the possibility of a divide-by-zero runtime error bringing the program to a halt.

Short-circuit evaluation doesn't only apply to the **AND** operator. A string of expressions **OR**ed together will be evaluated only until the first True value is encountered. More complicated expressions are simply ground through until the compiler is certain that nothing further can change the ultimate Boolean value. Then it will stop.

Using short-circuit Boolean evaluation is made easier by the fact that the compiler assumes it by default. What is called "complete" Boolean expression evaluation must be explicitly chosen if you want to use it. Forcing complete Boolean expression evaluation is done through the /B+ switch when using the command-line version of the compiler.

You can also force complete Boolean expression evaluation by inserting a **{$B+}** compiler command in your source code. By bracketing a region of code between a **{$B+}** command and a **{$B-}** command, you can force complete evaluation only between the two commands, and use short-circuit evaluation throughout the rest of your program.

Why would you ever want to use complete Boolean expression evaluation? Just as you sometimes need to ensure that a certain subexpression will *never* be evaluated (as we saw above), you must sometimes ensure that *every* subexpression *is always* evaluated.

Almost all such cases involve Pascal functions that return Boolean values after doing some sort of necessary work that must be completed regardless of which value the function returns. (For those of you reading this book serially who may not yet understand Pascal functions and how they return values, look ahead to Chapter 10, especially Section 10.1.)

```
IF AllocateBigBuffer(BigBuffPtr) AND
  AllocateLittleBuffer(LittleBuffPtr)
    THEN LoadBothBuffers ELSE
      BEGIN
        IF BuffPtr1 <> NIL THEN LoadBuffer1
          ELSE IF BuffPtr2 <> NIL THEN LoadBuffer2
       ELSE UseDiskSwap := True
      END;
```

This example comes from a program that needs a lot of memory for buffers. It attempts to allocate both its large and its small buffers if possible. If only the large buffer can be allocated without leaving enough RAM for the small buffer, so be it. Or, if the large buffer cannot be allocated but the small buffer can, the small buffer will be allocated and used. Finally, if the program can't find enough memory to allocate either RAM-based buffer, it will use a disk-swapping system to make disk space serve as (slower) memory space for buffer operations. **BuffPtr1** and **BuffPtr2** are pointers that are initialized to point to their buffers when those buffers are allocated. If there isn't enough memory to allocate a buffer, its pointer is returned with a value of **NIL** to indicate that its buffer could not be created.

Whether or not both buffers can be created in memory, *both* pointers must be initialized to some value, either a legitimate pointer value to an allocated buffer, or else NIL. Later on in the program, the logic will need to test those pointers to see if a given buffer exists. Having either pointer in an uninitialized state could be disastrous.

If we allowed short-circuit Boolean expression evaluation here, function **AllocateLittleBuffer** would never be executed if **AllocateBigBuffer** failed to find enough memory to allocate the large buffer. **LittleBuffPtr** would be left in an uninitialized state, and later on, the program could malfunction if it tried to test the uninitialized **LittleBuffPtr**.

In this situation, the entire Boolean expression

```
IF AllocateBigBuffer(BigBuffPtr) AND
  AllocateLittleBuffer(LittleBuffPtr)
```

has to be evaluated, regardless of the results of executing the function **AllocateBigBuffer**. The only way to guarantee this is to force complete Boolean expression evaluation, either by issuing a command to the IDE or to the command-line compiler, or by bracketing this area of code between **{$B+}** and **{$B-}** commands.

As the somewhat specialized and arcane nature of this example suggests, short-circuit Boolean evaluation will be your method of choice in the vast majority of instances.

9.3. CASE

Choosing between one of several alternative control paths is critical to computer programming. We've seen how **IF/THEN/ELSE** in its simplest form can choose between two alternatives based on a Boolean expression. By nesting **IF/THEN/ELSE** statements one within another, we can choose among many different control paths, as we saw in the previous section.

The problem of readability appears when we nest **IF** statements more than two or three deep. **Nested IF/THEN/ELSE** gets awkward and non-intuitive in a great hurry when more than three levels exist. Consider the problem of flashing a message on a display screen based on some input code number. A problem reporting system on a display-equipped car computer system might include a statement sequence like this:

```
Beep;
Writeln('*****WARNING*****');
IF ProblemCode = 1 THEN
  Writeln('[001] Fuel supply has fallen below 10%')
ELSE IF ProblemCode = 2 THEN
  Writeln('[002] Oil pressure has fallen below min spec')
ELSE IF ProblemCode = 3 THEN
  Writeln('[003] Engine temperature is too high')
ELSE IF ProblemCode = 4 THEN
  Writeln('[004] Battery voltage has fallen below min spec')
ELSE IF ProblemCode = 5 THEN
  Writeln('[005] Brake fluid level has fallen below min spec')
ELSE IF ProblemCode = 6 THEN
  Writeln('[006] Transmission fluid level has fallen below min spec')
ELSE IF ProblemCode = 7 THEN
  Writeln('[007] Radiator water level has fallen below min spec')
ELSE
  Writeln('[***] Logic failure in problem reporting system');
```

This will work well enough, but it takes some picking through to follow it clear to the bottom. This sort of selection of one statement from many based on a single selection value is what the **CASE** statement was created for. Rewriting the above statement with **CASE** gives us this:

```
Beep;
Writeln('*****WARNING*****');
CASE ProblemCode OF
  1 : Writeln('[001] Fuel supply has fallen below 10%');
  2 : Writeln('[002] Oil pressure has fallen below min spec');
  3 : Writeln('[003] Engine temperature is too high');
  4 : Writeln('[004] Battery voltage has fallen below min spec');
```

```
   5 : Writeln('[005] Brake fluid level has fallen below min spec');
   6 : Writeln('[006] Transmission fluid level is below min spec');
   7 : Writeln('[007] Radiator water level has fallen below min spec')
ELSE
   Writeln('[***] Logic failure in problem reporting system')
END; { CASE }
```

Here, **ProblemCode** is called the *case selector*. The case selector may be an expression or a variable. It holds the value upon which the choice among statements will be made. The numbers in a line beneath the word **CASE** are called *case labels*. Each case label is followed by a colon and a statement. The statement may be simple or compound.

When the **CASE** statement is executed, the case selector is evaluated and its value is compared, one by one, against each of the case labels. If a case label is found equal to the value of a case selector, the statement associated with that case label is executed. Once the statement chosen for execution has completed executing, the work of the **CASE** statement is done and control passes to the rest of the program. Only one (or none, see below) of the several statements is executed for each pass through the **CASE** statement.

Although my examples here show only single statements associated with case labels, compound statements are perfectly legal.

If no case label matches the value of the case selector, the statement following the **ELSE** is executed. **ELSE** is optional, by the way; if no **ELSE** is found, control falls through to the next statement in the program.

The general form of a **CASE** statement is this:

```
CASE <case selector> OF
   <constant list 1> : <statement 1>;
   <constant list 2> : <statement 2>;
   <constant list 3> : <statement 3>;
                 ....
   <constant list n> : <statement n>
   ELSE <statement>
END;
```

There may be as many case labels as you like, up to 256. You may be puzzling over the fact that what we pointed out as case labels are called *constant lists* in the general form. In our first example, each case label was only a single numeric constant. A case label may also be a list of constants separated by commas. Remember that the case label is the *list* of constants associated with a statement; each statement can only have *one* case label. And do not forget that a case label may *never* be a variable!

For another example, let's look at some code for a mail-in survey analysis system. The responses must be grouped by geographical regions of the country. **State** is an

enumerated type including all the standard two-letter state name abbreviations, in alphabetical order. This particular code fragment tallies the number of responses from each geographical region:

```
TYPE
  {II = Indiana; OG = Oregon to avoid reserved word conflict}
  State = (AK,AL,AR,AZ,CA,CO,CT,DE,DC,FL,GA,HI,ID,IL,II,
           IA,KS,KY,LA,MA,MD,ME,MI,MN,MO,MS,MT,NE,NV,NH,
           NJ,NM,NY,NC,ND,OH,OK,OG,PA,RI,SC,SD,TN,TX,UT,
           VA,VT,WA,WI,WV,WY)

VAR
  FromState : State;

CASE FromState OF
CT,MA,ME,NH,
RI,VT          : CountNewEngland := CountNewEngland + 1;
DC,DE,MD,NJ,
NY,PA          : CountMidAtlantic := CountMidAtlantic + 1;
FL,GA,NC,SC    : CountSoutheast := CountSoutheast + 1;
IA,IL,II,MI,
MN,OH,WI,WV    : CountMidwest := CountMidwest + 1;
AL,AR,KY,LA,
MO,MS,TN,VA    : CountSouth := CountSouth + 1;
KS,ND,NE,SD,
WY             : CountPlains := CountPlains + 1;
AK,CA,CO,HI,
ID,MT,OG,UT,
WA             : CountWest := CountWest + 1;
AZ,NM,NV,OK,
TX             : CountSouthwest := CountSouthwest + 1;
END; { CASE }
```

Here you can see that a case label can indeed be a list of constants. Also note that there is no **ELSE** clause here because every one of the possible values of type **State** is present in one of the case labels.

CASE cautions

The most important thing to remember about case labels is that they must be constants or lists of constants. A particular value may appear only once in a **CASE** statement. In other words, the value **IL** (from the last example) could not appear in both the **CountMidwest** and the **CountSouth** case labels. The reason for this should be obvious; if a value is associated with more than one statement, the **CASE** logic will not know which statement to execute for that case label value.

You should be careful when using a case selector of type **Integer**. Case selectors may only have values between 0 and 255. An integer case selector may have a value

much larger than 255, and when it does the results of executing the **CASE** statement are undefined. If you work a lot with numeric codes (and intend to use **CASE** structures to interpret those codes) it's a good idea to define those codes as subranges of **Integer**:

```
TYPE
  Keypress = 0..255;
  Problem  = 0..32;
  Priority = 0..7;
```

Any of these named subrange types may act as case selectors. FreePascal and Delphi also provide type **Byte**, which is an unsigned 8-bit integer that can only take values from 0-255. For that reason, **Byte** makes the perfect case selector for numeric values.

ISO Standard Pascal lacks ELSE in CASE

I should point out an important variance between Turbo Pascal and ISO Standard Pascal here: One of the puzzling lapses of logic in ISO Standard Pascal is the lack of an **ELSE** clause in its definition of **CASE**. In Standard Pascal, a case selector value for which no case label exists is supposed to cause a run-time error. The programmer is supposed to ensure, with range testing, that each value submitted to a **CASE** statement is in fact legal for that **CASE** statement. No good explanation for why this should be necessary has ever crossed my desk.

It isn't surprising that every single commercial implementation of Pascal that I've ever tested includes an **ELSE** clause of some sort in its definition, to cover that "none of the above" possibility. Turbo Pascal uses the reserved word **ELSE** for this purpose. FreePascal supports the reserved word **OTHERWISE** as a synonym for **ELSE** in **CASE** statements, as did a number of other Pascal compilers (such as UCSD Pascal and the old Turbo Pascal for the Macintosh) but the meaning and function are exactly the same. Note well that **OTHERWISE** may *only* be used as a synonym for **ELSE** in **CASE** statements; don't try it in **IF/THEN/ELSE** statements.

9.4. FOR LOOPS

There are many occasions when you must perform the same operation or operations on a whole range of values. The most-used example would be the generation of a square roots table for all the numbers from 1 to 100. Pascal provides a tidy way to loop through the same code for each value, then drop through to the next statement in the program when all the loops have been performed. It's called the **FOR** statement, and it is one of three ways to perform program loops in Pascal.

Printing the table of square roots becomes easy:

```
FOR I := 1 TO 100 DO
  BEGIN
    J := Sqrt(I);
    Writeln('The square root of ',I,' is ',J)
  END;
```

There are better ways to lay out a square roots table, obviously, but this gets the feeling of a **FOR** loop across very well. **I** is an integer. **J** is a real number. The compound statement between the **BEGIN/END** pair is executed 100 times. The first time through, **I** has the value 1. Each time the compound statement is executed, the value of **I** is increased by 1. Finally, after the compound statement has been executed with the value of **I** as 100, the **FOR** statement has done its job and control passes on to the next statement in the program.

The preceding example is only a particular case of a **FOR** statement. The general form of a **FOR** statement is this:

```
FOR <control variable> :=
  <start value> TO <end value> DO <statement>
```

<start value> and <end value> may be expressions. <control variable> is any ordinal type, including enumerated types. When a **FOR** statement is executed, the following things happen: If <start value> and <end value> are expressions, they are evaluated and tucked away for reference. Then the control variable is assigned <start value>. Next, <statement> is executed. After <statement> is executed, the *successor value* to the value already in the control variable is placed in the control variable. The control variable is now tested. If it exceeds <end value>, execution of the **FOR** statement ceases. Otherwise <statement> is executed.

The loop repeats until the control variable is incremented past <end value.>

Note that the general definition of a **FOR** statement does not speak of "adding one to" the control variable. The control variable is incremented by assigning the successor value of the current value to the control variable. "Adding" is not really done at all, not even with integers. To obtain the successor value, the statement evaluates the expression

```
Succ(<control variable>)
```

The function **Succ** is discussed in more detail later on in Section 11.6. If you recall our enumerated type Spectrum:

```
TYPE
  Spectrum = (Red, Orange, Yellow, Green, Blue,
              Indigo, Violet);
```

the successor value to **Orange** is **Yellow**. The successor value to **Green** is **Blue**, and so on.

A variable of type **Spectrum** makes a perfectly good control variable in a **FOR** loop:

```
LightSpeed := 3.0E05;
FOR Color := Red TO Violet DO
  Frequency[Color] := Lightspeed / Wavelength[Color];
```

So do characters:

```
FOR Ch := 'a' to 'z' DO
  IF Ch IN CharSet DO Writeln('CharSet contains ',Ch);
```

If <start value> and <end value> are the same, the loop is executed once. If <start value> is higher than <end value>, the loop is not executed at all. That is, <statement> is not executed, and control immediately falls through to the next statement in the program.

Control Variable Cautions

A control variable must be an ordinal type or a subrange of an ordinal types. Real numbers *cannot* be used as control variables. There is no distinct successor value to a number like 3.141592, after all. For similar reasons you cannot use sets or structured types of any kind.

Also, an error message will appear if the control variable is a formal parameter passed by reference (see Section 10.3), that is, a **VAR** parameter in the function or procedure's parameter line. For example, you cannot do this:

```
PROCEDURE Runnerup(Hi,Lo     : Integer;
                   VAR Limit : Integer);

VAR
  Foo : Integer;

BEGIN
  <statements>;
  FOR Limit := Lo TO Hi DO <statement>;
  Foo:=Limit;
  <statements>
END;
```

FreePascal will respond here with an error message. Standard Pascal requires that control variables be local and non-formal. FreePascal is somewhat more lenient, and allows control variables to be non-local (this is, not declared in the current block; see Chapter X for more on this) and also allows formal parameters to be control variables as long as they are passed by value, that is, if the procedure is given its own copy of the parameter to play with.

As with most of Pascal's rules and restrictions, this one was designed to keep you out of certain kinds of trouble. Understanding what kind of trouble requires a little further poking at the notion of control variables in **FOR** loops: To make fault procedure **Runnerup** shown above work, some sort of local control variable would have to be declared in the data declaration section of the procedure, like this:

```
PROCEDURE Runnerup(Hi,Lo    : Integer;
                   VAR Limit : Integer);
VAR
  Foo,I : Integer;

BEGIN
  <statements>;
  FOR I := Lo TO Hi DO <statement>;
  Foo := I;
  <statements>
END;
```

Now the **FOR** loop will compile correctly. But there is still something wrong with this procedure. What it's trying to do is make use of the control variable immediately after the loop has executed by assigning its value to **Foo**. This is also illegal. *Immediately after a* **FOR** *statement, the value of the control variable becomes undefined.* This is not a problem, since the end value is accessible in **Hi**. To make use of the final value of the control variable, assign **Hi** to Foo instead of **I**. The end result will be the same:

```
PROCEDURE Runnerup(Hi,Lo    : Integer;
                   VAR Limit : Integer);


VAR
  Foo,I : Integer;

BEGIN
  <statements>;
  FOR I := Lo TO Hi DO <statement>;
  Foo := Hi;
  <statements>
END;
```

There is a good reason for the control variable becoming undefined after its loop has run its course. After each pass through <statement> the successor value to the current value in the control variable is computed. In the process, that successor value may in fact become undefined if the ordinal type being used runs out of values. Going back to our type **Spectrum**, what is the successor value to **Violet**? There is none; **Succ(Violet)** is undefined. Consider:

```
FOR Color := Red TO Violet DO <statement>;
```

After the loop runs through its seven iterations, **Color** would hold an undefined value. If you were allowed to "pick up" the control variable's value after the run of a **FOR** loop, you might in fact be picking up an undefined, nonsense value and have no way of knowing that it were so. This is why the Standard Pascal definition declares that control variables are *always* undefined after a **FOR** loop to remove the temptation to "save" the final value of a control variable for later use.

In Borland Pascal V7, a **Break** statement was added to the language. **Break** interrupts execution of a **FOR**, **WHILE**, or **REPEAT** loop before the loop has run through all the iterations called for. With **Break** now an option, you cannot always be sure that the <end value> will match the control variable's value at the end of the loop, so you must use a separate variable if you want to keep track of the value in the control variable and use it after the loop exits.

Within the **FOR** loop (that is, within <statement>) the control variable must be treated as a "read-only" value. You *can* change the value of the control variable within the FOR loop, but you should do so with extreme hesitation. The **REPEAT/UNTIL** and **WHILE/DO** statements are designed to support this kind of "moving target" loop, which will execute as often as required to make a control variable equal to some final value. **FOR** loops, by contrast, were designed to execute a fixed number of times, and while executing, the value of the control variable should be solely under the control of the **FOR** loop itself.

Now (finally!) you may understand why Pascal forbids using **VAR** parameters as control variables. Pascal reserves the right to force a control variable into an undefined state after its loop is done. Using a **VAR** parameter as a control variable might "reach up" out of the procedure and force the **VAR** parameter into some unexpected and possibly undefined state. Allowing a procedure to "undefine" a parameter passed to it is asking for trouble, since it may not be obvious to the calling logic that its parameter may come back undefined.

Preserve sanity in your programs. Keep your **FOR** loop control variables local.

FOR with DOWNTO

The **FOR** statement as we've seen it so far always increments its control variable "upward"; that is, it uses the successor value of the control variable for the next pass through <statement>. It is sometimes useful to go the other way: To begin with a "high" value and count down to a lower value. In this case, the predecessor value of the value in the control variable becomes the new control variable value for the next pass through <statement>. This predecessor value is calculated with the predefined function **Pred**(<control variable>); see Section 11.6. Otherwise, its operation is identical to that of **FOR** with **TO**:

```
FOR I := 17 DOWNTO 7 DO <statement>;

FOR Color := Indigo DOWNTO Orange DO <statement>;

FOR Ch := 'Z' DOWNTO 'X' DO <statement>;
```

When using **DOWNTO**, keep in mind that if <start value> is lower than <end value>, <statement> will not be executed at all. This is the reverse of the case for **FOR/TO** loops.

There's no STEP in FreePascal!

One thing to keep in mind when learning to use **FOR** loops: The loop can only "step through" the range the control variable must take one value at a time. Some languages (including some implementations of BASIC) allow a **STEP** parameter to specify that the loop is to "count by five" or some other value. Pascal does not allow you to do this. If you want the loop to count up or down by some value other than one, you need to either skip the "in-between" values while counting (the **MOD** operator can be handy for this) or else take the gutsy road and jigger the value of the control variable on each pass. This can be done, but be careful!

9.5. WHILE/DO LOOPS

As we've seen, a **FOR** loop executes a specific number of times, no more, no less, unless you use the **Break** or **Continue** procedures, as I'll describe in Section 9.X. The control variable should not be altered during the loop. There are many cases in which a loop must run until some condition occurs that stops it. The control variable *must* be altered during the loop, or the loop will just run forever. Pascal offers two ways to build such loops: **WHILE/DO** and **REPEAT/UNTIL**.

The general form of a **WHILE/DO** loop is this:

```
WHILE <Boolean expression> DO <statement.>
```

The <Boolean expression> can be an expression that evaluates to a Boolean value, such as **I > 17**, or it can simply be a Boolean variable. As with **FOR** loops, <statement> can be any statement, including a compound statement framed between **BEGIN** and **END**.

WHILE/DO loops work like this: The code first evaluates <Boolean expression>. If the value of the expression is **True**, then <statement> is executed. If the value is **False**, <statement> is not executed even once, and control falls through to the next statement in the program. (Don't forget that the **WHILE/DO** loop as a whole is considered a statement.)

Assuming that <Boolean expression> came out **True** the first time, then after executing <statement> the code goes back and evaluates <Boolean expression> again. If the expression is still **True**, <statement> is executed again. If it evaluates to **False**, the **WHILE/DO** loop ends and control passes on to the next statement in the program.

In short, as long as <Boolean expression> is **True**, <statement> will be executed repeatedly. Only when the expression comes up **False** will the loop end.

For example:

```
VAR
  pH : Real;

FillTank;            { Fill tank with raw water   }
Take(pH);            { Take initial PH reading     }
WHILE pH < 7.2 DO
  BEGIN
    AddAlkali;       { Drop 1 soda pellet in tank }
    AgitateTank;     { Stir }
    Take(pH);        { Read the PH sensor          }
  END;
```

This snippet of code is a part of a control system for some sort of chemical processing apparatus. All it must do is fill a tank with water, ensuring that the pH of the water is at least 7.2. If the water from the water supply comes in as too acidic (water quality varies widely in some parts of the country) its pH must be brought up to 7.2 before the water is considered useable.

First the tank is filled. Then the initial pH reading is taken. The water may in fact be useable from the start, in which case the loop is never executed. But if the water comes up acidic, the loop is executed. A small quantity of an alkali is added, the

tank is stirred for a while, and then the pH is taken again. If the pH has risen to 7.2, the loop terminates. If the pH remains too low, the loop is executed again, more alkali is added, and the pH is tested once more. This will continue until the pH test returns a value in the variable pH that is higher than 7.2.

It's crucial to note that an initial pH test was performed. If the variable **pH** were not used before, its value is undefined, and testing it for **True** or **False** will not have any real-world meaning. *Make sure the Boolean expression is defined before the* **WHILE/ DO** *loop is executed!* Every variable in the expression must be initialized somehow before the expression can be "trusted."

The most important property of a **WHILE/DO** loop is that its Boolean expression is tested *before* <statement> is executed. A corollary to this is that there are cases when <statement> will never be executed at all. Keep this in mind while we discuss **REPEAT/UNTIL** next.

9.6. REPEAT/UNTIL Loops

REPEAT/UNTIL loops are very similar to **WHILE/DO** loops. As with **WHILE/ DO**, **REPEAT/UNTIL** executes a statement until a Boolean expression becomes True. The general form is this:

```
REPEAT <statement> UNTIL <Boolean expression>;
```

It works this way: First, <statement> is executed. Then <Boolean expression> is evaluated. If it comes up **True**, the loop terminates, and control passes on to the next statement in the program. If <Boolean expression> evaluates to **False**, <statement> is executed again. This continues until <Boolean expression> becomes **True**.

The important fact to notice here is that <statement> is always executed at least once. So unlike **WHILE/DO**, you needn't initialize all variables in <Boolean expression> before the loop begins. It's quite all right to assign all values as part of executing the loop.

For an example, let's return to the chemical process controller and consider a snippet of code to handle a simple titration. Titration means adding small, carefully measured amounts of one chemical to another while watching for some chemical reaction to go to completion. Usually, when the reaction is complete, the mixture will begin changing color, or will become electrically conductive, or give some other measurable signal.

In Pascal, it might be handled this way:

```
VAR
  Drops    : Integer;
  Complete : Boolean;

Drops := 0;
REPEAT
  AddADrop;              { Opens valve for 1 drop   }
  Drops := Drops + 1;    { Increment counter        }
  Signal(Complete)       { Read the reaction sensor }
UNTIL Complete;
```

Note that the drops counter is initialized before the loop begins. A drop is added to the test vessel, and the drop counter is incremented by one. Then the reaction sensor is read. If it senses that the reaction has gone to completion, the Boolean variable **Complete** is set to **True**.

Since it takes it least one drop to complete the reaction, (without any drops of the chemical, the reaction can't even begin) this series of events must be done at least once. If the first drop completes the reaction, the loop is performed only once. Most likely, the loop will have to execute many times before the chemical reaction goes to completion and **Complete** becomes **True**. When this happens, **Drops** will contain the number of drops required to complete the reaction. The value of **Drops** might then be displayed on a display of some kind attached to the chemical apparatus.

One interesting thing about **REPEAT/UNTIL** is that the two keywords do double duty if <statement> is compound. Instead of bracketing the component statements between **BEGIN** and **END**, **REPEAT** and **UNTIL** perform the bracketing function themselves.

WHILE/DO or REPEAT/UNTIL?

These two types of loops are very similar, and it is sensible to ask why both are necessary. Actually, **WHILE/DO** can accomplish anything **REPEAT/UNTIL** can, with a little extra effort and occasional rearranging of the order of the statements contained in the loop. The titration code could be written this way:

```
Drops := 0;
Complete := False;
WHILE NOT Complete DO
  BEGIN
    AddADrop;
    Drops := Drops +1;
    Signal(Complete)
  END;
```

This method requires that **Complete** be set to **False** initially to ensure that it will be defined when the loop first tests it. If **Complete** were left undefined and it happened to contain garbage data that could be interpreted as the value **True**, (if bit 0 is binary 1, for example) the loop might never be executed, and the code would report that the titration had been accomplished with zero drops of reagent—which is chemically impossible.

Using **REPEAT/UNTIL** would prevent that sort of error in logic. Quite simply: *Use* **REPEAT/UNTIL** *in those cases where the loop must be executed at least once.* Whenever you write code with loops, always consider what might happen if the loop were never executed at all—and if anything unsavory might come of it, make sure the loop is coded as **REPEAT/UNTIL** rather than **WHILE/DO**.

9.7. LABELS, GOTO, BREAK, AND CONTINUE

The bane of ancient unstructured languages like BASIC, FORTRAN, and COBOL is freeform **GOTO** branching all over the program body without any sort of plan or structure. Such programs are very nearly impossible to read. The problem with such programs, however, is not the **GOTO**s themselves but bad use of them. **GOTO**s fill a certain (exceedingly rare) need in Pascal, but they have a seductive power and on the surface are eminently easy to understand. Just say

```
GOTO 150;
```

and *wham!* you're at the location marked by label 150. This straightforwardness leads inexperienced programmers (especially those first schooled in BASIC) to use them to get out of any programming spot that they do not fully understand how to deal with in a Pascal-style structured manner.

I will not caution you, as some people do, never to use a **GOTO** no matter what. What I will tell you to do is never use a **GOTO** when something else will get the job done as well or better.

Labels

In order to use **GOTO**, **GOTO** must have somewhere to go, and in Pascal that somewhere must be marked by something called a label. Labels, like most everything else in Pascal, must be predeclared before they are used. Declaring labels is done in the label declaration section of a program, immediately following the reserved word **LABEL:**

```
LABEL
  100,150,200,250,300;
```

This example **LABEL** definition statement conforms to ISO Standard Pascal. The labels themselves must be numeric for Standard Pascal, in the range 0-9999. A label may mark only *one* point in a program. That is, you may not mark two different locations in a single program with the same label.

Most modern Pascal implementations, including FreePascal, extend the label syntax and allows labels to be ordinary identifiers just like those you use for constants and variables. You can mix numeric and identifier labels in the same label definition statement:

```
LABEL
  100,ShutDown,PowerGlitch,200,300,HardwareFailure;
```

When you mark a statement with a label, you must put a colon after the label. Look to the code a few paragraphs down for an example.

Numeric and identifier labels are used identically in **GOTO** statements:

```
GOTO ShutDown;
GOTO 200;
```

The effect is the same in either case: Execution continues at the statement in the program marked by the label.

GOTO limitations in FreePascal

Use of **GOTO**s with FreePascal carries a few limitations. You may **GOTO** a label within the current block. This means that you may *not* **GOTO** a label inside another procedure or function, or from within a procedure or function out into the main program block. You should not **GOTO** a label within a structured statement. In other words, given this **WHILE/DO** statement:

```
WHILE NOT Finished DO
  BEGIN
    Read(MyFile,ALine);
    IF EOF(MyFile) THEN GOTO 300;
    IF ALine = 'Do not write this...' THEN GOTO 250;
    Write(YourFile,ALine);
    LineCounter := LineCounter + 1;
    250:
  END
300: Close(MyFile);
```

you could not, from some other part of the block, **GOTO** label **250**. However, assuming that this snippet of code is not part of some larger structured statement, you could in fact **GOTO** label **300** from some other part of the current block. Whether or not that would perform any useful function is a good question.

This example is very bad practice, but we include it here only to indicate that you cannot branch into the middle of a **WHILE/DO** statement from somewhere outside the statement. Label **250** is accessible only from somewhere between the **BEGIN** and **END** pair.

In general, **GOTO**s are used to get *out* of somewhere, not to get *in*. Standard Pascal and versions of Borland's Pascal compilers prior to V7.0 lack two general looping features that most Pascals now have: **BREAK** and **CONTINUE**. **BREAK** leaves the middle of a loop and sends control to the first statement *after* the loop. **CONTINUE** stops executing the loop and begins the loop *at the top*, with the next value of the control variable.

Without **BREAK** and **CONTINUE**, **GOTO** provides the only reasonably clean way to get out from inside certain very complicated loops in which a lot is going on and more than one condition value affects exiting from the loop. Logically speaking, you can always get out of a loop safely by using the facilities provided by the loop (exiting at the top with **WHILE/DO** and at the bottom with **REPEAT/UNTIL**) but there are cases when to do so involves tortuous combinations of **IF/THEN/ELSE** that might in fact be harder to read than simply jumping out with **GOTO**. But for clarity's sake, always **GOTO** the statement immediately following the loop you're exiting. (This is precisely what the **BREAK** statement does.)

BREAK and **CONTINUE** are now in the Pascal language definition, as I'll describe below in more detail. That being the case, the possible situations requiring **GOTO** have dwindled down to practically nothing.

Such situations are rare enough that FreePascal forbids the use of **GOTO** by default. In order to use **GOTO**, you'll have to place the **{$GOTO ON}** compiler switch in your source code before you use a **GOTO** statement. The **{GOTO OFF}** switch turns off **GOTO** permissions, and is assumed when you run the compiler until a **{$GOTO ON}** is encountered.

With **BREAK** and **CONTINUE** in hand, what remains is the very rare need to get somewhere else right *now*, especially when the code you need to get to is code to handle some impending failure or emergency situation that requires immediate attention to accomplish an orderly shutdown of the equipment, or something of similar seriousness. This would tend to come up in embedded-system type software that has direct control over some rather complicated hardware. In thirty-five years of working with Pascal I have never had to do this. I suspect that if you ever do, you'll know it.

Break and Continue

As I mentioned earlier, modern Pascal compilers provide the **BREAK** and **CONTINUE** statements. Both statements may only be used in connection with looping statements **FOR**, **WHILE/DO**, and **REPEAT/UNTIL**. **BREAK** may also be used in **CASE** statements, but **CONTINUE** may not.

BREAK amounts to a **GOTO** that takes execution to the statement that immediately follows the loop statement. Consider:

```
Total := 0;
TotalAveraged := 0;
FOR I := 1 TO DataTally DO
  BEGIN
    IF (Total+DataPoints[I]) >= MaxLongInt THEN Break;
    Total := Total + DataPoints[I];
    Inc(TotalAveraged);
  END;
WriteDataToFile;
```

Here, a **FOR** loop is totalling data values stored in an array, for averaging. A statement has been added to prevent the total from overflowing a long integer variable. If the total plus the next data value is greater than **MaxLongInt** (a built-in constant containing the largest legal **LongInt** value) the **BREAK** statement is executed. **BREAK** takes execution to the first statement after the end of the **FOR** loop's compound statement; in this example, to the **WriteDataFile** procedure call. It's an "early exit" from the **FOR** loop, and that's about all to be said for it.

One caution: Don't count on the value of the **FOR** loop's control variable being available and meaningful once you leave the loop via **BREAK**. In my example, I'm keeping a separate count of the number of items successfully tallied, in the variable **TotalAveraged**. That's always a good idea, even if it seems to complicate the logic.

The use of **BREAK** within **CASE** is simple to describe: **BREAK** ends execution of the statements of an individual case before that case finishes. Remember that a statement associated with a case label may be a compound statement containing several simple statements. Sometimes you may want to just call a case done and exit the **CASE** statement as a whole. **BREAK** will do that, and in some cases may be simpler to understand than wrapping statements inside multiple **IF/THEN/ELSE** constructs.

CONTINUE is a little subtler. It's a "restart" procedure that allows you to "short circuit" the remaining logic in a loop, and start again at the first statement inside the loop. If the loop is a **FOR** loop, the control variable is bumped to its next value before the loop is restarted. If the loop is a **WHILE/DO** or **REPEAT/UNTIL** loop, the loop is restarted at the top, but no variables are affected in any way.

The following example isn't the best possible coding practice, but it's simple and shows what the **CONTINUE** statement actually does. The example reads a line from a text file and tests to see if there's anything in the line. That is, is tests to see if the length of the line read from the file is 0. If the line contains data and has a nonzero length, the **AddLineToList** procedure adds the line to a linked list.

```
WHILE NOT EOF(InFile) DO
BEGIN
  Readln(InFile,InLine);
  IF Length(InLine) <=0 THEN Continue ELSE
    AddLineToList(InLine);
END;
```

On the other hand, if the length of the line read from the text file is zero, there's really no more work to be done with that line anyway. Restart the loop from the top, which reads the next line from the file. Yes, again: This is an unnecessary complication, but it illustrates what **CONTINUE** does.

The real situations where **CONTINUE** is likely to be useful are complex loops that do a lot of things and test a lot of different conditions. Such loops make poor examples in tutorial books like this because their complexity confuses what they're really trying to demonstrate.

There can be multiple **BREAK** or **CONTINUE** statements inside the same loop. Be careful, however, that you don't find yourself using the two statements carelessly, as an excuse not to think through a loop's logic. **BREAK** and **CONTINUE** are used more often than **GOTO**—but not so often that you'll have them in every loop you write.

9.8. Semicolons and Where They Go

Nothing makes newcomers to Pascal cry out in frustration quite so consistently as the question of semicolons and where they go. There are places where semicolons must go, places where it seems not to matter whether they go or not, and places where they cannot go without triggering an error. Worse, it seems at first to have no sensible method to it.

Of course, like everything else in Pascal, placing semicolons does have a method to it. Why the confusion? Two reasons:

1. Pascal is a "freeform" language that does not take line structure of the source file into account. Unfortunately, a lot of new Pascal programmers graduate into Pascal from BASIC, which is about as line-oriented a language as ever existed. Also, one of the most popular modern programming languages, Python, is not freeform. What BASIC no longer impresses on new programmers, Python often does.

2. Semicolons in Pascal are statement *separators,* not statement *terminators.* The difference is crucial, and made worse by the fact that the C language family and its antecedents like PL/1 use semicolons as statement terminators.

Clarifying these two issues should make Pascal semicolon placement second nature.

Freeform vs. line-structured source code

Pascal source code is "freeform;" that is, the boundaries of individual lines and the positioning of keywords and variables on those lines matter not at all. The prettyprinting customary to Pascal source code baffled me in my earliest learning days until I realized that the compiler completely ignored it. The compiler, in fact, sucks the program up from disk storage as though through a drinking straw, in one long line. The following two program listings are utterly identical as far as a Pascal compiler is concerned:

```
PROGRAM Squares;

VAR
  I,J : Integer;

BEGIN
  Writeln('Number     Its square');
  FOR I := 1 TO 10 DO
    BEGIN
      J := I * I;
      Writeln('  ',I:2,'          ',J:3)
    END;
  Writeln;
  Writeln('Processing completed!')
END.
```

```
PROGRAM Squares;VAR I,J : Integer;BEGIN Writeln
('Number     Its square');FOR I:=1 TO 10 DO
BEGIN J:=I*I;Writeln('  ',I:2,'         ',J:3)
END;Writeln;Writeln('Processing completed!') END.
```

Although the second listing appears to exist in four lines, this is only for the convenience of the printed page; the intent was to express the program as one continuous line without any line breaks at all.

The second listing above is the compiler's eye view of your program source code. You must remember that although you see your program listing "from a height" as it were, the compiler scans it one character at a time, beginning with the 'P' in

PROGRAM and reading through to the "." after **END**. All unnecessary "whitespace" characters (spaces, tabs, carriage returns, linefeeds) have been removed as the compiler would remove them. Whitespace serves only to delineate the beginnings and endings of reserved words and identifiers, and as far as the compiler is concerned, one whitespace character of any kind is as good as one of any other kind. Once the compiler "grabs" a word or identifier, literal or operator, it tosses out any following whitespace until it finds a non-whitespace character indicating that a program element is beginning again.

Semicolons as statement separators

Note the compound statement executed as part of the FOR loop:

```
BEGIN J:=I*I;Writeln(' ',I:2,'          ',J:3) END
```

There are two statements here, framed between **BEGIN** and **END**. Smart as the FreePascal compiler may seem to you, it has no way to know where statements start and end unless you tell it somehow. If the ';' between **I** and **Writeln** were not there, the compiler would not know for sure if the statement that it sees (so far!) as **J:=I*I** ends there or must somehow continue on with **Writeln**.

Note that there is no semicolon after the second statement. There doesn't have to be; the compiler has scanned a **BEGIN** word and knows that an **END** should be coming up eventually. The **END** word tells the compiler unambiguously that the previous statement is over and done with. **BEGIN** and **END** are *not* statements. They are reserved words, acting as delimiters, and only serve to tell the compiler that the group of statements between them is a compound statement.

It's useful to think of a long line of statements as a line of boxcars on a rail siding. Separating each car from the next is a pair of linked couplers. Anywhere two couplers connect is where, (if boxcars were program statements) you would need a semicolon. You don't need one at the front of the first car, or at the end of the last car because the last car doesn't need to be separated from anything; behind it is just empty air.

The null statement

Why, then, is it legal to have a semicolon after the last statement in a compound statement? This is perfectly all right (and adds to the confusion):

```
BEGIN
  J := J + 5;
  IF J > 100 THEN PageEject;
  DoPage; { ; legal but not needed here }
END
```

The answer, of course, is that there **is** a statement after statement **DoPage**; and that statement is the null statement. This might be clearer with the example rewritten this way:

```
BEGIN
  J := J + 5;
  IF J > 100 THEN PageEject;
  DoPage;
  { Null statement here! }
END
```

There is a semicolon between **DoPage** and the null statement, but none between the null statement and the **END** word.

The null statement is in truth a theoretical abstraction. It doesn't really exist the same way an **IF** statement exists. It does no work and generates no code, not even a **NOP** (No-Op) assembly language instruction. It serves very little purpose other than to make certain conditional statements a little more intuitive and readable. For example:

```
IF TapeIsMounted THEN { NULL } ELSE RequestMount;
```

I find this more readable than the alternative:

```
IF NOT TapeIsMounted THEN RequestMount;
```

but I suspect it's a matter of taste. Note my convention (which I've seen elsewhere) of inserting the comment **{ NULL }** wherever you use a null statement. It's like the bandages around the Invisible Man; they make the guy easier to see and thus keep him out of trouble.

Another use of the null statement is in **CASE** statements in which nothing needs to be done for a specific selector value:

```
CASE Color OF
  Red    : { NULL };          { No filter needed }
  Orange : InsertFilter(1);   { Density 1        }
  Yellow : InsertFilter(5);   { Density 5        }
  Green  : InsertFilter(11);  { Density 11       }
  ELSE InsertFilter(99)       { Opaque (99)      }
END; { CASE }
```

In some sort of optical apparatus there is a mechanism for rotating a filter in front of an optical path. The density of the filter depends on the color of light being used. No filter is needed for red, and for blue, indigo, or violet the test will not function and an opaque barrier is moved into the optical path instead of a filter. A null statement is used for the **Red** case label.

Semicolons with IF/THEN/ELSE statements

More errors are made placing semicolons within **IF/THEN/ELSE** statements than any other kind, I suspect. This sort of thing is fairly common and oh, so easy to do when you're a beginner:

```
IF TankIsEmpty THEN FillTank(Reagent,FlowRate);
  ELSE Titrate(SensorNum,Temp,Drops);
```

The temptation to put a semicolon at the end of a line is strong. Furthermore, in most dialects of BASIC you must put a colon between an IF clause and its associated **ELSE** clause.

But semicolons are statement separators, and the example above is one single statement. There is nothing to separate. Remember this rule with regard to placing semicolons in **IF/THEN/ELSE** statements: *Never* place a semicolon immediately before an **ELSE** word in an **IF** statement! With that in mind you will avoid 90% of all semicolon placement errors.

To make things slightly more confusing, it is legal (and sometimes necessary) to place a semicolon before an **ELSE** word in a **CASE** statement. Given that FreePascal allows the use of the **OTHERWISE** reserved word as an alias for **ELSE** within a **CASE** statement, it becomes possible to simply say, never place a semicolon immediately before **ELSE**. In the meantime, keep the null statement in mind and semicolon placement won't seem quite so arbitrary.

Forward reference: EXIT, Error, and exception handling

I'm ending this chapter without including a discussion of the **EXIT** statement. It's about getting out of functions and procedures before they've finished executing, just as **BREAK** is about getting out of loops before they've finished executing. The next chapter discusses procedures and functions in detail; **EXIT** will be there as well.

There is one additional topic related to structuring code: error and exception handling. It's possible to write code in FreePascal that gracefully recovers from various sorts of runtime errors. Key to exception handling are the **TRY/FINALLY** and **TRY/EXCEPT** statements. These will have to wait until the second book I'm working on. In the meantime, go take a look at the FreePascal Wiki's coverage of exceptions:

```
https://wiki.freepascal.org/Exceptions
```

Chapter 10.
Procedures and
Functions

Some people think that conditional and looping statements like those we studied in the previous chapter are the touchstone of structured programming. Not so: At the bottom of it all, *structured programming is the artful hiding of details.* The human mind's ability to grasp complexity breaks down quickly unless some structure or pattern can be found in the complexity. I recall (with some embarrassment) writing a 600-line APL program in 1979, and by the time I wrote the last of it (this being done over a six week period) I no longer remembered how the first part worked. The entire program was a mass of unstructured, undifferentiated detail. Those who have dabbled in APL may lay a little of the blame on APL; most of it I lay on myself.

How does one hide details in computer programs? By identifying sequences of code that do discrete tasks, and setting each sequence off somewhere, replacing it by a single word describing (or at least hinting at) the task it does. Such code sequences are properly called "subprograms."

10.1. Procedures vs. Functions

In Pascal, there are two types of subprograms: *procedures* and *functions*. Both are sequences of Pascal statements set off from the main body of program code. Both are invoked, and their statements executed, simply by naming them. The only difference between functions and procedures is this: The identifier naming a function has a type and takes on a value when it is executed. The name of a procedure has no type and takes on no value.

Two simple examples: The procedure **ClrScr**, when executed, clears the screen of a console text window. The very simple console programs I presented in the first part of this book, like Aliens.pas, used it:

```
Clrscr;
```

ClrScr is a complete statement. Although **ClrScr** has no parameters, a procedure may have any number of parameters if it needs them. More on this shortly.

245

Functions

A function, by contrast, is *not* a complete statement. It is more like an expression, which returns a value that must be used somehow:

```
VAR
 Space,Radius : Real;

Radius := 4.66;
Space := Area(Radius);
```

The **Area** function calculates the area for the value **Radius**, which is passed to it as a parameter. After it calculates the area, the area value is taken on by the identifier **Area**, as though **Area** were a variable.

Functions in Pascal may return values of just about any predefined or custom type. This is a *huge* difference from earlier Pascal compilers, most of which could return only simple types and strings.

Using the **Area** function hides the details of calculating areas. There aren't many details involved in calculating areas, but for other calculations (matrix inversion comes to mind) a function can hide thirty or forty lines of complicated code, or (often) a lot more. So when you're reading the program and come to a function invocation, you can think, "Ah, here's where we invert the matrix" without being concerned about *how* the matrix is actually inverted. At that level in reading the program, the how is not important, so those details are best kept out of sight.

"Throwing away" function results with extended syntax

FreePascal supports an extension to Pascal syntax, allowing you to "throw away" the value returned by a function, thus invoking the function as though it were a procedure. The function result is always calculated inside the function body, but the function does not have to deliver the result by residing within an expression or on the right side of an assignment statement.

This doesn't make sense for most functions that you write, since in most cases, the return value is the whole purpose for writing the function to begin with. However, as your functions grow more complex, you might on occasion wish to invoke a function simply for its "side effects"—that is, the things it does that don't involve the return value.

This happens most often when you create a function that does something significant like file access, and returns a code as its function return value that tells whether or not the action was a success. In most cases, you'll want to check the function result to see how it went. There may, however, be an occasion where you simply want to do the deed regardless of the status value. So instead of coding it this way:

```
FileActionStatus := CloseTheFile;
```

you could simply code it this way:

```
CloseTheFile;
```

FreePascal allows functions to be invoked this way by default. If you're a function purist, you can turn off this "extended syntax" by placing a **{$X-}** command somewhere in your source code file before you call the function. This can be useful for spotting "naked" function invocations while you're porting Pascal code to a compiler that doesn't support extended syntax.

The Exit Statement

FreePascal, like Turbo Pascal, offers the **Exit** statement. **Exit** jumps out of the current block into the next highest block. In other words, if you execute an **Exit** within a function or procedure, execution of that function or procedure will cease, and control returns to the statement immediately following the invocation of the function or procedure. If **Exit** is encountered in the main program, FreePascal will end the program and return to the IDE or the command line. With FreePascal it's possible for **Exit** to take a parameter. This allows a function to return a value to its caller even if **Exit** ends the function in the middle somewhere.

Use **Exit** with care. One of the strengths of the Pascal structure is the assurance that a block of code begins at the top and ends at the bottom. Sprinkling a block with **Exit** statements makes code much harder to read and debug.

The structure of functions and procedures

Procedures and functions are, in effect, miniature programs. They can have their own label declarations, constant declarations, type declarations, variable declarations, and procedure and function declarations, as well as all the expected code statements. Consider these two entities:

```
PROGRAM HiThere;         PROCEDURE HiThere;


BEGIN                    BEGIN
  Writeln('Hi there!')     Writeln('Hi there!')
END.                     END;
```

The only *essential* differences between a program and a procedure are the reserved word **PROGRAM** and the punctuation after the final **END**.

Functions are a little different. A function has a type and takes on a value that it returns to the program logic that invokes it:

```
FUNCTION Area(R : Real) : Real;

CONST
  PI = 3.14159;

BEGIN
  Area := PI * R * R;
END;
```

The type of function **Area** is **Real**. Inside the function, an expression computing area for the given radius **R** is evaluated, and the value is assigned to the function's name. Aside from these two distinctions, functions are identical to procedures and are also miniature programs.

10.2. FORMAL AND ACTUAL PARAMETERS

Passing data to procedures and functions can be done two ways: By using global variables that any procedure or function can read from or write to, and through each procedure or function's parameter list. The first method (sometimes called "common," from an old FORTRAN scheme) is generally a bad idea. It's far better practice to hand a procedure or function everything it needs through its parameter list.

The following program contains a procedure to draw boxes in the text console window with characters, as opposed to true pixel graphics. In this age of ubiquitous GUIs in our operating systems, this may seem less than useful, but it's a good example of procedures and functions in action.

```
{----------------------------------------------------------}
{                         BoxTest                          }
{                                                          }
{ Character box-draw demo program; demos concept of procedures }
{                                                          }
{                           by Jeff Duntemann             }
{                           FreePascal 3.0                 }
{                           Last update 10/29/2016         }
{                                                          }
{    From: FREEPASCAL FROM SQUARE ONE  by Jeff Duntemann    }
{----------------------------------------------------------}

PROGRAM BoxTest;

USES Crt;
```

```
TYPE
  LineRec = RECORD
                  ULCorner,
                  URCorner,
                  LLCorner,
                  LRCorner,
                  HBar,
                  VBar,
                  LineCross,
                  TDown,
                  TUp,
                  TRight,
                  TLeft : String[4]
               END;


CONST
  PCLineChars : LineRec =
    (ULCorner : #201;
     URCorner : #187;
     LLCorner : #200;
     LRCorner : #188;
     HBar     : #205;
     VBar     : #186;
     LineCross: #206;
     TDown    : #203;
     TUp      : #202;
     TRight   : #185;
     TLeft    : #204);



VAR
  X,Y           : Integer;
  Width,Height : Integer;



PROCEDURE MakeBox(X,Y,Width,Height : Integer;
                  LineChars        : LineRec);


VAR
  I : Integer;

BEGIN
  IF X < 0 THEN X := (80-Width) DIV 2;    { Negative X centers box }
  WITH LineChars DO
    BEGIN                                 { Draw top line }
      GotoXY(X,Y); Write(ULCorner);
      FOR I := 3 TO Width DO Write(HBar);
      Write(URCorner);
                                          { Draw bottom line }
```

```
      GotoXY(X,(Y+Height)-1); Write(LLCorner);
      FOR I := 3 TO Width DO Write(HBar);
      Write(LRCorner);
                                          { Draw sides }
      FOR I := 1 TO Height-2 DO
        BEGIN
          GotoXY(X,Y+I); Write(VBar);
          GotoXY((X+Width)-1,Y+I); Write(VBar)
        END
    END
END;



BEGIN
  Randomize;                  { Seed the pseudorandom number generator }
  ClrScr;                     { Clear the entire text window }
  WHILE NOT KeyPressed DO     { Draw boxes until any key is pressed }
    BEGIN
      X := Random(72);        { Get a Random X/Y for UL Corner of box }
      Y := Random(21);
      REPEAT Width := Random(80-72) UNTIL Width > 1;  { Get Random Height &}
      REPEAT Height := Random(25-Y) UNTIL Height > 1; { Width to fit on the }
      MakeBox(X,Y,Width,Height,PCLineChars);          { display & draw it! }
      Delay(25);
    END
END.
```

The **PCLineChars** typed constant is an excellent example of the use of record constants, as I described them in the last chapter. But what I wanted this little program to demonstrate is the use of a procedure to hide some program details.

Procedure **MakeBox** has a parameter list with five parameters in it. In the parameter list of the procedure's declaration they are named: **X**, **Y**, **Width**, **Height**, and **LineChars**. Notice that the types of these parameters are given in the procedure declaration. **X**, **Y**, **Width**, and **Height** are all identical types, so they may be given as a list separated by commas. You could also have defined each of the four separately, like this:

```
PROCEDURE MakeBox(X         : Integer;
                  Y         : Integer;
                  Width     : Integer;
                  Height    : Integer;
                  LineChars : LineRec);
```

The parameters defined in a procedure's declaration are called *formal parameters* and must always be given a type, separated from the formal parameter by a colon. In the example, **X**, **Y**, **Width**, **Height**, and **LineChars** are all formal parameters.

When a procedure is invoked, values are passed to the procedure through its

parameters. The compiler understands the parameter types from reading the procedure declaration. Types are not given in the invocation:

```
MakeBox(25,BoxNum+2,30,3,PCLineChars);
```

Furthermore, in this example, the values may be values stored in variables, or values expressed as constants or expressions. The parameters that are present in the parameter list of a particular invocation of a procedure are called **actual parameters**. (All parameters passed to **MakeBox** are passed *by value*. If they were passed by *reference*, the actual parameters would have to be variables of identical type to the formal parameters. This will be fully explained in the next section.)

Identifiers used as formal parameters are local to their procedure or function. As such, their names may be identical to identifiers defined in other procedures or functions, or in the main program, without any conflict. (This is about *scope*, and I'll explain scope at the end of this chapter.) The **X** and **Y** formal parameters in **MakeBox** have no relation at all to an **X** or **Y** identifier used elsewhere in the program. Of course, the flipside of this is also true: If you are using an **X** or **Y** variable global to the entire program they will not be accessible from within **MakeBox**. If you try to access a global **X** or **Y** variable from within **MakeBox**, you will access the formal parameters **X** and **Y** instead.

Also remember that formal **VAR** parameters may not act as control variables in **FOR** loops. Local variables (such as **I** in **MakeBox**) should be declared for this purpose. Non-**VAR** formal parameters may be used freely as control variables in **FOR** loops.

10.3. Passing Parameters By Value Or By Reference

When a function or a procedure is invoked, the actual parameters are "meshed" with the formal parameters, and then the function or procedure does its work. The meshing of actual parameters with formal parameters is done two ways: by value, and by reference.

Passing parameters by value

A parameter passed by value is just that: a value is copied from the actual parameter into the formal parameter. The movement of the value to the procedure is a one-way street. Nothing can come back out again and be used by the calling program. This applies whether the actual parameter is a constant, an expression, or a variable.

There are powerful advantages to one-way data movement *into* a procedure. The procedure can fold, spindle, and mutilate the parameter any way it needs to, and not

fear any side effects outside of the procedure. The copy of the actual parameter it gets is a truly private copy, strictly local to the procedure itself.

If a variable is passed to a procedure by value the type of the variable must be compatible with the type of the formal parameter.

Passing parameters by reference

There are many occasions when the whole point of passing a parameter to a function or procedure is to have it modified and returned for further use. To have a procedure or function modify a parameter and return it, the parameter must be passed by reference.

Unlike parameters passed by value, a parameter passed by reference (often called a **VAR** parameter) cannot be a literal, a constant or an expression. The values of constants and literals by definition cannot be changed, and the notion of changing the value of an expression and stuffing it back into the expression makes no logical sense.

To be passed by reference, an actual parameter must be a variable of the *identical* type as the formal parameter. Compatible types will not do; the types must evaluate down to the same type definition statement.

The one exception to this rule in FreePascal involves short strings. Short strings, if you recall from Section 8.6, may be defined in any physical length from 1 to 255, with the default length being 255 unless you specify some other length. Under strict type checking a **VAR** string parameter passed to a procedure must be of the identical type declared in the procedure's header:

```
VAR
  String1 : String80;
  String2 : String30;

PROCEDURE DoSomething(VAR WorkString : String255);
```

In this example, strict type checking would prohibit passing either **String1** or **String2** as a parameter to procedure **DoSomething**.

However, FreePascal supports *relaxed type checking*, which would allow a **ShortString** of any length to be passed in the **WorkString** parameter. Relaxed type checking is the default. Strict vs. relaxed type checking is controlled with the **{$V}** compiler command. The long form is **{$VARSTRINGCHECKS}**. You must explicitly use the **{$V+}** command to impose strict type checking if desired. To use relaxed type checking and allow strings of any physical length to be passed as **VAR** parameters regardless of the formal **VAR** parameter's physical length, use **{$V-}**.

The draconian nature of strict type checking for **VAR** parameters makes a little more sense when you realize that the variable itself is not copied into the formal parameter (as with parameters passed by value). What is passed is actually a pointer to the variable itself. Data is not being moved from one variable to another. Data is being read from one variable and written back into the same variable. To protect other data items that may exist to either side of the variable passed by reference, the compiler insists on a *perfect* match between formal and actual parameters.

There is a cost to relaxed type checking: A string actual parameter that is too long to fit into its VAR formal parameter will be truncated to the length of the **VAR** formal parameter. No warning will appear, and doing so will lose any character data beyond the length of the formal parameter.

The **{$V}** compiler switch is provided mostly as for compatibility with older versions of Turbo Pascal. Borland Pascal V7 provided a new feature, *open string parameters*, that does the same thing a lot more safely. FreePascal supports open string parameters as well. I'll discuss them a little later in this chapter.

To illustrate parameter passing, let's look at a more sophisticated procedure than we've seen so far. The **MakeBox** procedure I described earlier had several parameters, all passed by value. For an example of a parameter passed by reference, consider the shell sort procedure below. Note that this is a procedure, not a complete program. I'll present a complete sort demonstration program that incorporates this procedure a little later in this chapter.

```
{->>>>ShellSort<<<<------------------------------------------}
{                                                            }
{ Filename : SHELSORT.SRC -- Last Modified 10/30/2016        }
{                                                            }
{ This is your textbook Shell sort on an array of key records, }
{ defined as the type shown below:                           }
{                                                            }
{     KeyRec = RECORD                                        }
{                 Ref     : Integer;                         }
{                 KeyData : String30                         }
{              END;                                          }
{                                                            }
{     From: FREEPASCAL FROM SQUARE ONE  by Jeff Duntemann    }
{------------------------------------------------------------}

PROCEDURE ShellSort(VAR SortBuf : KeyArray; Recs : Integer);

VAR
  I,J,K   : Integer;
  Spread  : Integer;
```

```pascal
PROCEDURE KeySwap(VAR RR,SS : KeyRec);

VAR
  T : KeyRec;

BEGIN
  T := RR;
  RR := SS;
  SS := T
END;


BEGIN
  Spread := Recs DIV 2;         { First Spread is half record count  }
  WHILE Spread > 0 DO           { Do until Spread goes to zero:       }
    BEGIN
      FOR I := Spread + 1 TO Recs DO
        BEGIN
          J := I - Spread;
          WHILE J > 0 DO
            BEGIN                   { Test & swap across the array }
              K := J + Spread;
              IF SortBuf[J].KeyData <= SortBuf[K].KeyData THEN J := 0 ELSE
                KeySwap(SortBuf[J],SortBuf[K]);
              J := J - Spread
            END
        END;
      Spread := Spread DIV 2    { Halve Spread for next pass }
    END
END;
```

This procedure sorts an array of sort keys. A *sort key* is a record type that consists of a piece of data and a pointer to a file entry from which the data came. The fastest and safest way to sort a file is not to sort the file at all, but to build an array of sort keys from information in the file and sort the array of sort keys instead. The array can then be written out to a file. Since the data in the array is in sorted (usually alphabetical) order, it can be searched using a fast binary search function. Once a match to a desired string is found (in the **Key** field of a **KeyRec** record) the **RecNum** field contains the physical record number of the record in the file where the rest of the information is stored.

I should point out here that although this was the traditional way to construct simple databases in Pascal, modern compilers like FreePascal and environments like Lazarus provide direct access to real database engines like MySQL and SQLite, so constructing your own databases and sorting key files is no longer necessary for writing useful data handling software.

Now, look at the parameter line for **ShellSort**:

```
PROCEDURE ShellSort(VAR SortBuf : KeyArray; Recs : Integer);
```

The first parameter, **SortBuf**, is passed by reference. The second parameter, **Recs**, is passed by value. The difference is that **SortBuf** is preceded by the keyword **VAR**. **VAR** indicates that the parameter following it is passed by reference.

The reason for passing **SortBuf** by reference should be obvious: We want to rearrange the sort keys in **SortBuf** and put them in a certain order. **ShellSort** does this rearranging. The code calling **ShellSort** will need to get **SortBuf** "back" when the rearranging is done. Had we passed **SortBuf** to **ShellSort** by value, **ShellSort** would have received its own private copy of **SortBuf**, would have sorted the copy, and then would have had no way to return the sorted copy to the rest of the program.

Recs contains a count of the number of sort keys loaded into the array **SortBuf**. While knowing the value stored in **Recs** is essential to sorting **SortBuf** correctly, it need not be changed, and thus Recs can be passed by value. Only the value of **Recs** is needed.

Summing up:

- An actual parameter passed by value is copied into the formal parameter. The copy is local to the procedure or function and changes made to the copy do not "leak out" into the rest of the program.

- Passing a parameter to a procedure by reference actually gives the procedure a pointer to the physical variable being passed. Changes made to the parameter within the procedure are actually made to the physical variable outside the procedure.

- To pass a parameter by reference, precede the parameter by the keyword VAR. When passed by reference, actual parameters must be variables of *identical type* to the formal parameter.

10.4. Open Array and String Parameters

One of the hassles of working with Pascal arrays is that Pascal will not allow you to pass an array to a procedure in a formal array parameter that does not match the actual array parameter in every respect, including element type, upper bound, and lower bound. This makes it difficult to create a general-purpose array-sorting procedure, for example, since you must define in the array formal parameter exactly how many elements will be in the array passed into the procedure.

Borland Pascal v7 added *open array parameters* to address this problem, and FreePascal implements the feature in the current day. Veterans of programming

language theory and other Pascal compilers will recognize this feature as what Niklaus Wirth calls *conformant arrays.*

In the header of your procedure or function, a formal array parameter is declared with the type of the elements in the array, but without any declared bounds. We might declare a new version of the **ShellSort** procedure presented earlier in this chapter this way:

```
PROCEDURE ShellSort(VAR SortBuf : ARRAY of KeyRec; Recs : Integer);
```

The new **SortBuf** parameter is an open array parameter. We are told that it is an array of **KeyRec**, but not how *large* an array SortBuf is. FreePascal is flexible enough to be able to handle the meshing of the formal array parameter **SortBuf** with an actual array of **KeyRec** at runtime. Either of the array variables below can be passed to the new **ShellSort** procedure in the **SortBuf** open array parameter:

```
VAR
  BigBuffer : ARRAY[0..500] OF KeyRec;
  LilBuffer : ARRAY[0..100] OF KeyRec;
```

No other changes need to be made to **ShellSort** to allow it work perfectly well with any size array of **KeyRec** passed to it, assuming that the **Recs** value accurately reflects the number of significant elements in the array passed. If **Recs** is larger than the upper bound of the array passed in the open array parameter, a range error will be triggered.

Now, inside of the new **ShellSort**, how does the procedure know what it's dealing with? To manipulate the arrays passed to it, the procedure must know what any array's upper and lower bounds are. The answer lies in two predefined functions **High** and **Low**. (Make sure before using open array parameters that you have not defined your own identifiers with the names **High** and **Low**.) Inside of a procedure or function having an open array parameter, **High(<open array>)** returns contains the value of the upper array bound of the actual array parameter, and **Low(<open array>)** returns the value of the lower bound of the actual array parameter.

High and **Low** are *only* valid within the procedure having the parameter they refer to. Don't try to refer to them globally, or from within some other procedure or function that does not itself have an open array parameter.

The **Recs** parameter of the **ShellSort** procedure exists to allow a partially-filled array of records to be sorted—**Recs** simply tells **ShellSort** how many elements of the **SortBuf** array really contain data. We can build some safety into **ShellSort** by using **High** to check whether **Recs** accidentally contains a larger number than

the upper bound of the **SortBuf** array. In that case, we can set **Recs** equal to the high bound of **SortBuf**, so that **ShellSort** will find itself passed an array that might be partly full, or full, but not *over*-full. (In an over-full array, records in excess of **High(SortBuf)** are ignored.) It only takes one simple statement, the first in the revised **ShellSort** shown below.

```
{ ShellSort implemented with open arrays.}

PROCEDURE ShellSort(VAR SortBuf : ARRAY OF KeyRec; Recs : Integer);

VAR
  I,J,K,L : Integer;
  Spread  : Integer;


PROCEDURE KeySwap(VAR RR,SS : KeyRec);

VAR
  T : KeyRec;

BEGIN
  T := RR;
  RR := SS;
  SS := T
END;



BEGIN
  { First we make sure Recs isn't higher than the upper bound: }
  IF Recs > High(SortBuf) THEN Recs := High(SortBuf);
  Spread := Recs DIV 2;         { First Spread is half record count  }
  WHILE Spread > 0 DO           { Do until Spread goes to zero:      }
    BEGIN
      FOR I := Spread + 1 TO Recs DO
        BEGIN
          J := I - Spread;
          WHILE J > 0 DO
            BEGIN               { Test & swap across the array }
              L := J + Spread;
              IF SortBuf[J].KeyData <= SortBuf[L].KeyData THEN J := 0 ELSE
                KeySwap(SortBuf[J],SortBuf[L]);
              J := J - Spread
            END
        END;
      Spread := Spread DIV 2    { Halve Spread for next pass }
    END
END;
```

There are some restrictions on open array parameters. You cannot assign to an open array parameter in its entirety; that is, you cannot treat the array as a whole in any way. You can only work with an open array parameter on an element-by-element basis. Open array parameters passed by value (that is, without the **VAR** reserved word) are allocated on the stack, and can crash your machine if they take more stack than you've allocated for stack use.

Open string parameters

Open string parameters use the same machinery as open array parameters to allow you to pass strings of different sizes through a formal parameter without a specified string size. A string, after all, is simply an array of **Char** that receives special treatment from the runtime library in a few limited ways.

You declare an open string parameter using the predefined identifier **OpenString**. Here's a simple procedure that forces the case of its open string parameter to upper. You can pass a string value of any legal **ShortString** length (that is, up to 255 characters) in **Target** without running afoul of strong type checking:

```
PROCEDURE UCString(VAR Target : OpenString);

VAR
  I : Integer;

BEGIN
  FOR I := 1 TO Length(Target) DO
    Target[I] := UpCase(Target[I]);
END;
```

Inside the procedure, the standard **Length** string function works the way it does on any sort of string. **High(Target)** would return the defined length of the actual parameter passed in **Target. Low(Target)** will always return 0, since strings are always zero-based arrays.

Somewhat oddly, the identifier **OpenString** will not act this way when it is used to declare a value parameter (that is, without **VAR**.) As a value parameter, **OpenString** yields a string parameter that is always of type **STRING**, that is, the maximum string length of 255, and the **High** function will always return 255. You should only use **OpenString** as a **VAR** parameter!

Open short strings with $P+

Borland Pascal v7 introduced a new compiler command, **{$P+}**, as an alternate way to declare open string parameters, and FreePascal supports it. If you place the **{$P+}**

command toward the top of your source code file, **VAR** parameters declared using the **STRING** reserved word act as open string parameters, and not simply as strings with a maximum length of 255. However, value parameters of type **STRING** remain type **STRING** and do not become open array parameters. As with **OpenString**, the **VAR** has to be there! This was done to provide backward compatibility to older Turbo Pascal code that used **STRING** as a way of safely passing strings of any size to a procedure or function. If you're writing new code, the advised method is to use the **OpenString** predefined type instead.

Or not: Note well that the **{$P}** switch works only for short strings. It has no effect on code making use of **ANSIStrings**. The better path today is to use old-style short strings for compatibility with old code only, and use **ANSIString**s for all new work.

10.5. Recursion

Recursion is one of those peculiar concepts that seems to defy understanding totally, and depend completely on mystery for its operation, until eventually some small spark of understanding happens, and then, *wham!* It becomes simple or even obvious. A great many people have trouble understanding recursion at first glance, so if you do too, don't think less of yourself for it. For the beginner recursion is simple. But it is *not* obvious.

Recursion is what we call it when a function or procedure invokes itself. It seems somehow intuitive to beginners that having a procedure call itself is either impossible or else an invitation to disaster. Both of these fears are unfounded, of course. Let's look at them both.

Recursion is indeed possible. In fact, having a procedure call itself is no different from a coding perspective as having a procedure call any other procedure. What happens when a procedure calls another procedure? Only this: First, the called procedure is "instantiated;" that is, its formal parameters and local variables are allocated on the system stack. Next, the return address (the location in the code from which the procedure was called and to which it must return control) is "pushed" onto the system stack. Finally, control is passed to the called procedure's code.

When the called procedure is finished executing, it retrieves the return address from the system stack and then clears its variables and formal parameters off the stack by a process we call "popping." Then it returns control to the code that called it by branching to the return address.

None of this changes when a procedure calls itself. Upon a recursive call to itself, new copies of the procedure's formal parameters and local variables are instantiated on the stack. Then control is passed to the start of the procedure again.

The problem shows up when execution reaches the point in the procedure where it calls itself. A third instance of the procedure is allocated on the stack, and the procedure begins running again. A fourth instance, and a fifth…and after a few hundred recursive calls the stack has grown so large that it collides with something important in memory, and the system crashes. If you had this kind of procedure, such a thing would happen very quickly:

```
PROCEDURE Fatal;

BEGIN
  Fatal
END;
```

Such a situation is a sort of unlimited software feedback loop. It's this possibility that makes newcomers feel uneasy about recursion.

Obviously, *the important part of recursion is knowing when to stop.*

A recursive procedure must test some condition before it calls itself, to see if still needs to call itself in order to complete its work. This condition could be a comparison of a counter against a predetermined number of recursive calls, or some Boolean condition that becomes true (or false) when the time is right to stop recursing and go home. When controlled in this way, recursion becomes a very powerful and elegant way to solve certain programming problems.

Let's go through a simpleminded example of a controlled recursive procedure. Read through this program's code very carefully:

```
PROGRAM PushPop;

CONST
  Levels = 5;

VAR
  Depth : Integer;

PROCEDURE Dive(VAR Depth : Integer);

BEGIN
  Writeln('Push!');
  Writeln('Our depth is now: ',Depth);
  Depth := Depth +1;
  IF Depth <= Levels THEN Dive(Depth);
  Writeln('Pop!');
END;
```

```
BEGIN
  Depth := 1;
  Dive(Depth);
  Write('Press any key to exit: ');
  Readln;
END.
```

The program itself is nothing more than setting a counter to one and calling the recursive procedure **Dive**. Note the constant named **Levels**. **Dive** displays the word "Push!" when it begins executing, and the word "Pop!" when it ceases executing. In between, it displays the value of the variable **Depth** and then increments it.

If, at this point, the value of **Depth** is less than the constant **Levels**, **Dive** calls itself. Each call to **Dive** increments **Depth** by one, until at last **Depth** is greater than **Levels**. Then recursion stops.

Running program **PushPop** produces the output below. Can you tell yourself exactly why? (Note that the keypress prompt is not included on this page for simplicity's sake.)

```
Push!
Our depth is now 1
Push!
Our depth is now 2
Push!
Our depth is now 3
Push!
Our depth is now 4
Push!
Our depth is now 5
Pop!
Pop!
Pop!
Pop!
Pop!
```

Follow the execution of **PushPop** through, with a pencil to touch each keyword, if necessary, until the output makes sense to you.

10.6. Appplications of Recursion

Certain workaday programming problems simply cry out for recursive solutions. Perhaps the simplest and best-known is the matter of calculating factorials. A factorial is the product of a digit and all the digits less than it, down to one:

```
5! = 5 * 4 * 3 * 2 * 1
```

A little scrutiny here will show that 5! is the same as 5 * 4!, and 4! is the same as 4 * 3!, and so on. In the general case, N! = N * (N-1)! Whether you see it immediately or not, we have already expressed the factorial algorithm recursively by defining it in terms of a factorial. This will become a little clearer when we express it in Pascal:

```
FUNCTION Factorial(N : LongInt) : LongInt;

BEGIN
  IF N > 1 THEN Factorial := N * Factorial(N-1)
    ELSE Factorial := 1
END;
```

And that's it. We express it as a conditional statement because there must always be something to tell the code when to stop recursing. Without the **N > 1** test the function would merrily decrement **N** down past zero and recurse away until the system crashed.

The way to understand this function is to work it out for N=1, then N=2, N=3, and so on. For N=1 the N > 1 test returns False, so is assigned the value 1. No recursion involved. 1! = 1. For N=2 a recursive call to **Factorial** is made: **Factorial** is assigned the value **2 * Factorial(1)**. As we saw above, **Factorial(1) = 1**. So 2! = 2 * 1, or 2. For N=3, two recursive calls are made: **Factorial** is assigned the value **3 * Factorial(2)**. **Factorial(2)** is computed (as we just saw) by evaluating (recursively) **2 * Factorial(1)**. And **Factorial(1)** is simply = 1. Catching on? One interesting thing to do is add (temporarily) a **Writeln** statement to **Factorial** that displays the value of N at the beginning of each invocation.

A sidenote on the power of factorials: Calculating anything over 7! will overflow a two-byte integer. This is why the **Factorial** function returns a **LongInt** parameter. Here's an interesting exercise for you: How high a value can you pass in **N** without overflowing a long integer?

A recursive quicksort procedure

A considerably more useful application of recursion lies in the "quicksort" method of sorting arrays, invented by C.A.R. Hoare. Quicksort procedures can be written in a number of different ways, but the simplest way is by using recursion.

The quicksort procedure below does the same job that the procedure **ShellSort** did in the last section. **QuickSort** is passed an array of **KeyRec** and a count of the number of records to be sorted in the array. It rearranges the records until they are in ascending sort order in the array:

```
{->>>>QuickSort<<<<------------------------------------------}
{                                                            }
{ Filename : QUIKSORT.SRC -- Last Modified 10/30/2016        }
{                                                            }
{ This is your textbook recursive quicksort on an array of key }
{ records, which are defined as the type show below:         }
{                                                            }
{     KeyRec = RECORD                                        }
{                  Ref     : Integer;                        }
{                  KeyData : String30                        }
{                END;                                        }
{                                                            }
{     From: FREEPASCAL FROM SQUARE ONE  by Jeff Duntemann    }
{------------------------------------------------------------}

PROCEDURE QuickSort(VAR SortBuf : KeyARRAY;
                        Recs    : Integer);


PROCEDURE KeySwap(VAR RR,SS : KeyRec);

VAR
  T : KeyRec;

BEGIN
  T  := RR;
  RR := SS;
  SS := T
END;



PROCEDURE DoSort(Low, High : Integer);

VAR
  I,J   : Integer;
  Pivot : KeyRec;

BEGIN
  { Can't sort if Low is greater than or equal to High... }
  IF Low < High THEN
    BEGIN
      I := Low;
      J := High;
      Pivot := SortBuf[J];
      REPEAT
        WHILE (I < J) AND (SortBuf[I].KeyData <= Pivot.KeyData)
          DO I := I + 1;
        WHILE (J > I) AND (SortBuf[J].KeyData >= Pivot.KeyData)
          DO J := J - 1;
```

```
        IF I < J THEN KeySwap(SortBuf[I],SortBuf[J]);
      UNTIL I >= J;
      KeySwap(SortBuf[I],SortBuf[High]);
      IF (I - Low < High - I) THEN
        BEGIN
          DoSort(Low,I-1);     { Recursive calls to DoSort! }
          DoSort(I+1,High)
        END
      ELSE
        BEGIN
          DoSort(I+1,High);     { Recursive calls to DoSort! }
          DoSort(Low,I-1)
        END
    END
END;


BEGIN
  DoSort(1,Recs);
END;  { QuickSort }
```

QuickSort's *modus operandi* is summarized in Figure 10.1. One of the elements is chosen arbitrarily (here it is the last element in the array) to be the "pivot value." The idea is to divide the array into two partitions such that all elements on one side of the partition are greater than the pivot value, and all elements on the other side of the partition are less than the pivot value.
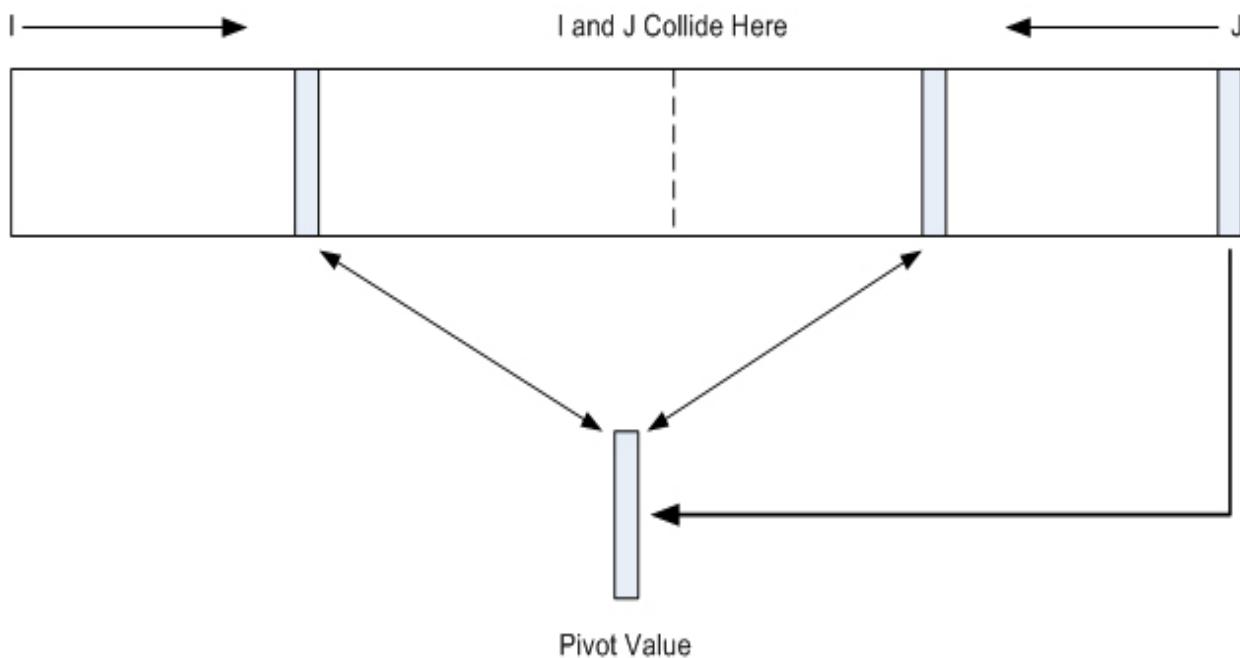

Figure 10.1. A QuickSort Scan

This is done by scanning the array from both ends toward the middle by counters **I** and **J**. **I** scans from the low end upward; **J** from the high end downward. The **I** counter samples each element, and stops when it finds an element whose value is higher than the pivot value. Then the scan begins from the top end down, with the J counter looking for a value that is less than the pivot value. When it finds one, the two found elements are swapped, thus putting them on the proper side of the pivot value.

When **I** and **J** collide in the middle somewhere (*not* necessarily in the center!) the array has been partitioned into two groups of elements: One that is larger than the pivot value, and one that is smaller than the pivot value. These two groups are not necessarily equal in size. In fact, they usually will *not* be. The only thing that is certain is that all the elements in one group are less than the value of the pivot element, and all of the elements of the other group are greater than the pivot element. The two groups are sorted with respect to one another: All elements of the low group are less than all elements of the high group.

Enter recursion: This same process is now applied to each of the two groups by calling **DoSort** recursively for each group. A new pivot value is chosen for each group, and each group is partitioned around its pivot value, just as the entire array was originally. When this is done, there are four groups. A little thought will show you that low-valued elements of the array are being driven toward the low end of the array, and high-valued elements are being driven toward the high end of the array. Within each group there is no guarantee that the elements are in sorted order. What you must understand is that *the groups themselves* are in sort order. In other words, all the elements of one group are greater than all the elements of the group below it.

Pressing on: Each of the four groups is partitioned again by more recursive calls to **DoSort**. The groups are smaller. Each group taken as one is sorted with respect to all other groups. With each recursive call, the groups have fewer and fewer members. In time, each group will contain only one element. Since groups are always in sort order, if each group is a single element, then all elements of the array are in sorted order, and **QuickSort**'s job is finished.

How does **QuickSort** know when to stop recursing? The first conditional test in **DoSort** does it: If **Low** is greater than or equal to **High**, the sort is finished. Why? Because **Low** and **High** are the bounds of the group being partitioned. If **Low** = **High**, each of the two groups has only one member. When the groups have only one member, the array is in sort order and the work is done.

If this makes your head spin, you're in good company. Follow it through a few times until it makes sense. Once you can follow **QuickSort**'s internal logic, you

will have a very good grasp of the uses of recursion.

This particular **Quicksort** algorithm works best when the original order of the elements in the array is random or nearly so. It works least well when the original order is close to fully sorted. For an array of random elements, it is one of the fastest of all sorting methods. For sorting arrays that are close to being in order, the **ShellSort** procedure given earlier will be consistently faster.

The following program puts the two sort procedures to the test. It generates a file of random keys, and allows you to display the random keys so that you can see how random they are. Finally, it will sort the file by whichever of the two methods you choose. Once the file has been sorted, you can display the keys once more to be sure that they have in fact been sorted.

The **SortTest** program uses a lot of concepts that I haven't introduced in the book yet, and calls routines that I haven't described so far. It is, however, a very straightforward program, and it would be an interesting exercise to see just how much of it you can understand based on what you've learned about Pascal so far.

```
{------------------------------------------------------------}
{                        SortTest                            }
{                                                            }
{             Data sort demonstration program                }
{                                                            }
{                           by Jeff Duntemann                }
{                           FreePascal V3.0                  }
{                           Last update 10/30/2016           }
{                                                            }
{     From: FREEPASCAL FROM SQUARE ONE  by Jeff Duntemann    }
{------------------------------------------------------------}


PROGRAM SortTest;

USES CRT,DOS,    { Standard Borland units }
     BoxStuff;  { Unit for drawing boxes; see Chapter 13 }

CONST
  Highlite   = True;  { These first 4 constants are used by WriteAt }
  NoHighlite = False;
  NoCR       = False;
  Shell      = True;  { Which sort procedure will we be using? }
  Quick      = False;


TYPE
  String30   = STRING[30];

  KeyRec     = RECORD
```

```
                     Ref     : Integer;
                     KeyData : String30
                  END;

  KeyArray  = ARRAY[0..500] OF KeyRec;
  KeyFile   = FILE OF KeyRec;

VAR
  IVal       : Integer;   { Holds integer value for user's response }
  WorkArray  : KeyArray;
  Randoms    : KeyFile;   { Files should generally be declared global }


{$I PULL.SRC }      { Described in Section 16.12 }
{$I CLREGION.SRC}   { Described in Section 18.1 }
{$I WRITEAT.SRC}    { Described in Section 18.3 }
{$I SHELSORT.SRC}   { Described in Section 14.2 }
{$I QUIKSORT.SRC}   { Described in Section 14.4 }


PROCEDURE GenerateRandomKeyFile(KeyQuantity : Integer);

VAR WorkKey : KeyRec;
    I,J     : Word;

BEGIN
  Assign(Randoms,'randoms.key');
  Rewrite(Randoms);
  FOR I := 1 TO KeyQuantity DO
    BEGIN
      FillChar(WorkKey,SizeOf(WorkKey),0);
      FOR J := 1 TO SizeOf(WorkKey.KeyData)-1 DO
        WorkKey.KeyData[J] := Chr(Pull(65,90));
      WorkKey.KeyData[0] := Chr(30);
      Write(Randoms,WorkKey);
    END;
  Close(Randoms)
END;


PROCEDURE DisplayKeys;

VAR WorkKey : KeyRec;

BEGIN
  Assign(Randoms,'randoms.key');
  Reset(Randoms);
  Window(25,13,70,22);
  GotoXY(1,1);
  WHILE NOT EOF(Randoms) DO
```

```
    BEGIN
      Read(Randoms,WorkKey);
      IF NOT EOF(Randoms) THEN Writeln(WorkKey.KeyData)
    END;
  Close(Randoms);
  Writeln;
  Writeln('        >>Press (CR)<<');
  Readln;
  ClrScr;
  Window(1,1,80,25)
END;


PROCEDURE DoSort(Shell : Boolean);

VAR I,Counter : Word;

BEGIN
  Assign(Randoms,'randoms.key');
  Reset(Randoms);
  Counter := 1;
  WriteAt(20,15,NoHighlite,NoCR,'Loading...');
  WHILE NOT EOF(Randoms) DO
    BEGIN
      Read(Randoms,WorkArray[Counter]);
      Counter := Succ(Counter)
    END;
  Close(Randoms);
  Write('...sorting...');
  IF Shell THEN ShellSort(WorkArray,Counter-1)
    ELSE QuickSort(WorkArray,Counter-1);
  Write('...writing...');
  Rewrite(Randoms);
  FOR I := 1 TO Counter-1 DO Write(Randoms,WorkArray[I]);
  Close(Randoms);
  Writeln('...done!');
  WriteAt(-1,21,NoHighlite,NoCR,'>>Press (CR)<<');
  Readln;
  ClearRegion(2,15,77,22)
END;

BEGIN
  ClrScr;
  MakeBox(1,1,80,24,PCLineChars);
  WriteAt(18,3,HighLite,NoCR,'FreePascal From Square One Sort Demo');
  REPEAT
    WriteAt(25,5,NoHighlite,NoCR,'[1] Generate file of random keys');
    WriteAt(25,6,NoHighlite,NoCR,'[2] Display file of random keys');
    WriteAt(25,7,NoHighlite,NoCR,'[3] Sort file via Shell sort');
    WriteAt(25,8,NoHighlite,NoCR,'[4] Sort file via Quicksort');
```

```
      WriteAt(18,10,NoHighlite,NoCR,'Enter 1-4, or 0 to quit, and press Enter:
');
   IVal := 0;
   Readln(IVal);
   CASE IVal OF
      0 :;                            { Null statement here; note semicolon }
      1 : GenerateRandomKeyFile(500); { 500 is as high as you can go...}
      2 : DisplayKeys;
      3 : DoSort(Shell);
      4 : DoSort(Quick);
      ELSE IVal := 0;
   END; {CASE}
  UNTIL (IVal = 0);
END.
```

10.7. FORWARD DECLARATIONS

In almost all cases, you must declare a type in the type declaration section of your program before you can declare a variable of that type. There's one well-known exception to this rule that I can't cover in detail in this book, as it involves the difficult issue of pointers, which will have to wait until my next book: You can use an identifier in a pointer definition before that identifier is defined. In other words, this pair of type definitions is completely legal:

```
TYPE
  RecPtr = ^DynaRec;
  DynaRec = RECORD
     DataPart : String;
     Next     : RecPtr
           END;
```

Ignore for now what **^DynaRec** means if you haven't been exposed to pointer notation before. What's significant here is that the FreePascal compiler "takes our word" that we will, in fact, define **DynaRec** before the program ends, and thus allows the definition of **RecPtr** before the definition of **DynaRec**—even though **DynaRec** is an integral part of the definition of **RecPtr**. Using an undefined identifier before its declaration is called a *forward reference*.

The context of defining pointer types is just about the only context in which Pascal will accept a forward reference to a type. (It's also true of object classes, which I won't be covering in this introductory volume.) In certain circumstances, however, Pascal can be persuaded to accept a procedure or function identifier before that procedure or function has been defined. In a sense, we must promise the compiler that we will, in fact, define the identifier. Or, if you prefer, we have to declare that we

will declare such an identifier somewhere down the source code trail.

This promise is called a *forward declaration*. It is accomplished with a Pascal reserved word, **FORWARD**, and it is done this way:

```
PROCEDURE NotThereYet(Foo,Bar : Integer); FORWARD;
```

What we have here is the procedure header all by itself, without any procedure body or local declarations of constants, types, or variables. People who understand units (See Chapter X) will think that this resembles the declarations in the interface section of a unit, and they are right.

Later on in the program, sometime before the **BEGIN** that marks the start of the main program block, procedure **NotThereYet** must be declared in its entirety. If it isn't, the compiler will hand us an error.

The eventual declaration of the procedure is perfectly ordinary. No special syntax indicates that the procedure had earlier been declared as **FORWARD**.

You do have the option of not re-declaring the forward declared procedure's parameter list. In other words, you could in fact define an (empty) procedure **NotThereYet** without parameters:

```
PROCEDURE NotThereYet;

BEGIN
END;
```

This shorthand, also, will be familiar to people who understand units. I consider it bad practice, however. The compiler allows you to re-declare the parameters, and re-declaring parameters contributes to the clarity of the program to have the parameter list in both places: The forward declaration, (where they are essential) and the full definition.

At last we come to the question: *What good is all of this?* The answer is: Not much. The only situation that genuinely requires forward declaration is circular (also called mutual) recursion. Consider these two procedure definitions:

```
PROCEDURE Egg;

BEGIN
  .
  .
  Chicken;
  .
  .
END;
```

```
PROCEDURE Chicken;

BEGIN
  .
  .
  Egg;
  .
  .
END;
```

Which comes first? You can't declare **Chicken** without calling **Egg**, and you can't declare **Egg** without calling **Chicken**. Pascal will call foul on the whole thing unless you forward declare one or the other. Adding the forward declaration makes everything work:

```
PROCEDURE Chicken; FORWARD;

PROCEDURE Egg;

BEGIN
  .            { Other program logic keeps this from }
  .            { being an infinite loop. }
  Chicken;   { Without the FORWARD, we get an error here. }
  .
  .
END;


PROCEDURE Chicken;

BEGIN
  .
  .
  Egg;
  .
  .
END;
```

That's what circular recursion *is*, but what it's good for has thus far escaped me. My hunch is that any program that appears to require circular recursion can probably be rewritten a different way without it.

I look upon the reserved word **FORWARD** much as I do the spokeshave sitting in the bottom drawer of my tool cabinet. I have never yet used it, but by golly, if I ever need to shave some spokes, I know just where it is.

Chapter 11.
Standard Functions

The ISO Standard Pascal definition includes a number of "standard functions" that are built into the language and need not be declared and coded into your program. These functions fall into two basic groups: Mathematical functions, which provide fundamental operations such as square and square root, absolute value, natural logarithms, and trig functions; and transfer functions, which define relationships between otherwise incompatible data types like **Integer** and **Char**.

Some books on Pascal refer to the parameter passed to a standard function as its "argument." This borrows jargon from the world of mathematics and may be confusing, since some people end up wondering what the difference is between an argument and a parameter. There is no difference in a Pascal language context. To lessen the confusion, I will use the term "parameter", which we have been using with respect to functions all along.

All of the standard functions described in this chapter may accept expressions as parameters, as long as those expressions evaluate to a value of the correct type. In other words, you may say **Sqrt(Sqr(X)+Sqr(Y))** as well as **Sqrt(16)**. Just make sure you don't try to extract the square root of a Boolean value, or of an enumerated type, and so on.

FreePascal implements all the standard functions from ISO Pascal, and quite a few of its own. In this chapter we'll discuss them in detail.

11.1. Round and Trunc

Round and **Trunc** are fence-sitters. They are both mathematical functions, in the sense that they provide a mathematical service, and they are also transfer functions, in that they provide a bridge between the partly-incompatible types **Real** and **Integer**.

We have already seen, in Chapter 7, that any integer value may be assigned to a variable of type **Real**. (This assumes that **Real** has sufficient range to express the

integer value.) But the reverse is not true, since a **Real** value may have a decimal part, and there is no way to express a decimal part in type **Integer**. **Round** and **Trunc** give us our choice of two ways to "transfer" a **Real** value into an **Integer** value. **Round** and **Trunc** both accept parameters of type **Real** and return values that may be assigned to either type **Integer** or **Real**.

Round

In mathematics, rounding a number means moving its value to the nearest integer. This is the job done by **Round**. **Round(X)** returns an integer value that is the integer closest to **X**. The direction in which a real number with a fractional part is rounded is usually given as "up" or "down". This can be confusing when you start dealing with negative real numbers. I prefer to visualize a number line and speak of "toward zero" or "away from zero."

- •   For **X** greater than 0: Rounds away from zero (up) for fractional parts greater than or equal to .5. Rounds toward zero (down) for fractional parts less than .5.

- •   For **X** less than 0: Rounds away from zero (down) for fractional parts greater than or equal to .5. Rounds toward zero (up) for fractional parts less than .5.

Some examples:

```
Round(4.449)        { Returns 4  }
Round(-6.12)        { Returns -6 }
Round(0.6)          { Returns 1  }
Round(-3.5)         { Returns -4 }
Round(17.5)         { Returns 18 }
```

Because the way rounding works with **Round** is symmetric with respect to zero, **Round(-X)** is equal to -**Round(X)**.

When **Round** is used with a parameter that is precisely halfway between two consecutive integers (for example, 3.5) the function uses *banker's rounding,* which rounds to the nearest even number. **Round** would take the parameter 3.5 and return 4; it would take the value 4.5 and also return 4.

Note that using **Round(X**) for a real value **X** that cannot be expressed as a long integer will generate a range check runtime error. This will occur if range checking is enabled; if range checking is not enabled, your program will continue executing, but the value actually assigned to the integer will be unpredictable, but predictably meaningless.

Trunc

Truncating a real number simply means removing its fractional part and dealing with what's left. **Trunc(X)** returns the closest integer value toward zero—and if you ponder that for a moment you'll see that it is equivalent to removing the fractional part and calling the whole number part an integer. Examples:

```
Trunc(17.667)        { Returns 17 }
Trunc(-3.14)         { Returns -3 }
Trunc(6.5)           { Returns 6  }
Trunc(-229.00884)    { Returns -229 }
```

Trunc returns a value of type **LongInt**. There were various range issues connected with **Trunc** in early versions of Turbo Pascal, relating to the lack of a **LongInt** type in Turbo Pascal 3.0 and earlier. (Long integers first appeared in Turbo Pascal 4.0.) Assuming the type to which you assign the value returned by **Round** or **Trunc** has the range to contain the value, there will be no problems with FreePascal.

11.2. SQR AND SQRT

Nothing complicated here. **Sqr(X)** squares **X**. It is completely equivalent to **X * X**, and Pascal includes it because squaring is done so frequently in mathematics, and also because (as we will discuss later) there is no exponentiation operator in Standard Pascal and hence no clean notation for **X** raised to a power of two. FreePascal *does* have an exponentiation operator, as I'll explain shortly, and so with FreePascal, **Sqr** is unnecessary.

Sqr may operate on both integers and reals. If you square an integer with **Sqr**, the returned value is an integer. If you square a real with **Sqr**, the returned value is real. Like **Sqr**, **Sqrt(X)** may operate on either an integer or real **X**. However (and this is important!) the value returned is *always* type **Real**.

A few examples:

```
VAR
  I : Integer;
  R : Real;

I := 64;
R := 6.077;

Sqrt(16)        { Returns 4.0; a real number! }
Sqrt(PI)        { Returns 1.77245; PI is a predefined constant }
Sqrt(I)         { Returns 8; real number }
Sqr(2.4)        { Returns 5.76; again, real }
```

```
Sqr(7)          { Returns 49; integer or real }
Sqr(I)          { Returns 4096; integer }
Sqr(R)          { Returns 36.92993; real }
```

The following procedure calculates the length of the hypotenuse of a right triangle, given the other two sides:

```
FUNCTION Hypotenuse(Side1,Side2 : Real ) : Real;

BEGIN
  Hypotenuse := Sqrt(Sqr(Side1) + Sqr(Side2))
END;
```

The algorithm here, of course, is the Pythagorean Theorem.

11.3. TRIGONOMETRIC FUNCTIONS

There are three trigonometric functions among the standard functions of Standard Pascal: **Sin**, **Cos**, and **ArcTan**. (The other trigonometric functions reside in the **Math** unit, as explained below.) **Sin(X)**, **Cos(X)**, and **ArcTan(X)** all return results of type **Real**. **X** may, however, be an integer or a real number. Note well that for these functions **X** represents radians, not degrees. A radian equals 57.29578 degrees. Radians, however, are usually thought of in terms of pi (3.14159) and fractions of pi. 360 degrees = 2pi radians; 180 degrees = pi radians, and so on. There is a standard function **Pi** that returns the value of pi. Pascal's trigonometric functions behave as you would expect them to behave from textbook discussions of trigonometry.

Trigonometric functions in the Math unit

Although all the other trig functions can be derived programmatically using the **Sin**, **Cos**, and **Arctan** functions, FreePascal provides a number of other precompiled trig functions in the **Math** unit.

ArcCos(X)	Computes the arc cosine of an angle
ArCosH(X)	Computes the hyperbolic arc cosine of an angle
ArcSin(X)	Computes the arc sine of an angle
ArcSinH(X)	Computes the hyperbolic arc sine of an angle
ArcTan2(X,Y)	Computes the arc tangent of two numbers
ArcTanH(X)	Computes the hyperbolic arc tangent of an angle
Cosecant(X)	Calculate cosecant (secant complement)
CosH(X)	Computes the hyperbolic cosine of an angle

Cotan(X)	Return cotangent (tangent complement)
Secant (X)	Calculate secant
SinH(X)	Computes the hyperbolic sine of an angle
Tan(X)	Computes the tangent of an angle
TanH(X)	Computes the hyperbolic tangent of an angle

All of these additional trig functions reside in the **Math** unit, and to use them you must place **Math** in your **USES** statement.

Note that even though the **X** and **Y** parameters passed to FreePascal's trig functions are declared as type **Real**, the compiler will allow you to pass an integer-type literal or variable in **X** without error, and will treat the value as a real number without a fractional part during the calculations.

11.4. ABSOLUTE VALUE, NATURAL LOGS, AND EXPONENTS

Absolute value

Absolute value in mathematics is the distance of a number from zero. In practical terms, this means stripping the negative sign from a negative number and leaving a positive number alone. The Pascal function **Abs(X)** returns the absolute value of **X**. The parameter **X** may be type **Real** or **Integer**. The type of the returned value is the same as the type of **X**. For example:

```
Abs(-61)        { Returns 61; type Integer }
Abs(484)        { Returns 484; also Integer }
Abs(3.87)       { Returns 3.87; type Real }
Abs(-61.558)    { Returns 61.558; also Real }
```

The **Abs** function is actually a shorthand form of the following statement:

```
IF X < 0 THEN X := - X;
```

Natural logarithms

There are two Pascal standard functions that deal with natural logarithms. Natural logarithms are mathematical functions that turn on a remarkable irrational number named e, which, to six decimal places, is 2.718282. Explaining where e comes from, or explaining natural logarithms in detail, is outside the scope of this book. Do read up on them (in any senior high math text) if the concept is strange to you.

Exp(X) returns the exponential function for **X**. The parameter may be a real

number or an integer, but the returned value is always real. The exponential function raises e to the X power. Therefore, when you evaluate **Exp(X)**, what you are actually evaluating is e^X .

Ln(X) returns the natural logarithm (logarithm to the base e) of **X**. The parameter may be type **Integer** or **Real**, and the returned value, again, is always type **Real**. The sense of the **Ln(X)** function is the reverse of **Exp(X)**: Evaluating **Ln(X)** yields the exponent to which e must be raised to give **X**.

Exponentiation

Natural logarithms are the most arcane of Pascal's mathematical standard functions. They are most used in mathematics that many of us would consider "heavy." However, there is one use for which natural logarithms fill an enormous hole in Standard Pascal's definition: Exponentiation. Unlike nearly all other programming languages, Pascal contains no general function for raising **X** to the **Y**th power. (In FORTRAN the exponentiation operator is the double asterisk: **X**Y** raises **X** to the **Y**th power.) **Exp** and **Ln** allow us to create a Standard Pascal function that raises one number to a given power:

```
FUNCTION Power(Mantissa, Exponent : Real) : Real;


BEGIN
  Power := Exp(Ln(Mantissa)*Exponent)
END;
```

This almost certainly looks like magic unless you really understand how natural logarithms work. Two cautions: The result returned is type **Real**, not **Integer**. Also, do not pass a zero or negative value in **Mantissa**. A runtime error will be triggered. Reason: **Ln(X)** for a negative **X** is undefined!

But that's Standard Pascal. Starting with version 4, Delphi added a new function to its **Math** unit: **Power**. FreePascal also includes **Power** in its **Math** unit:

```
FUNCTION Power(Mantissa, Exponent: Real) : Real
```

This is basically a precompiled version of the function shown above. Remember that it's located in the **Math** unit, and to use it you must place **Math** in your **USES** statement.

And that's not the end of it. FreePascal implements an exponention operator as well as an exponentiation function. The operator is the double asterisk, the same one used by FORTRAN. As with the **Power** function, the exponentiation operator

is stored in the **Math** unit, and you must include **Math** in your program to use it. (Math is not included in **USES** by default.) With the exponentiation operator, you raise a value to a power this way:

```
I := J**4;
```

This statement raises the value in **J** to the fourth power, and stores the result in **I**. As you might imagine, if you raise a real number value to a power, the result must be returned to a variable of type **Real**.

As far as I know, the double asterisk operator for exponentiation is used in no other Pascal implementation but FreePascal.

11.5. ORD & CHR

The functions **Ord** and **Chr** are true transfer functions, providing you with a well-documented, "legal" pathway between the otherwise incompatible types **Integer** and **Char**. **Ord** actually provides the pathway between integers and *any* ordinal type—hence the name.

Ord

As its name suggests, **Ord(X)** deals with ordinal types. Ordinal types are those types that can be "enumerated;" that is, types with a fixed number of values in a well-defined order. **Ord(X)** returns the ordinal position (an integer) of the value **X** in its ordinal type. The sixty-sixth character in the ASCII character set is the capital letter 'A'. **Ord('A')** returns 65. The third color in our old friend type **Spectrum** is **Yellow**. **Ord(Yellow)** returns 2—remember (for both examples) that we start counting at 0!

Chr

Chr goes in the opposite direction from **Ord**: **Chr(X)** returns a character value corresponding to the **X**th character in the ASCII character set. (**X** here is an integer.) **Chr(65)** returns the capital 'A'. **Chr(66)** returns capital letter 'B', and so on. Don't try to pass **Chr** an integer value higher than 255. The results will be undefined, and probably garbage.

The most important use of **Chr(X)** is generating character values that are not expressed by any symbol that you can place between single quote marks. How do you put a line feed in quotes? Or worse yet, a bell character? You don't—you express them with **Chr** :

```
Chr(13)          { Returns ASCII carriage return (CR) }
Chr(7)           { Returns ASCII bell (BEL)   }
Chr(127)         { Returns ASCII delete (DEL) }
Chr(8)           { Returns ASCII backspace (BS) }
```

Chr allows you to return a character based on an integer expression. The procedure **CapsLock**, for example, uses **Ord** and **Chr** to translate a character into an integer, manipulate the integer, and then translate the integer back into a character:

```
PROCEDURE CapsLock(VAR Target : String);

VAR Lowercase : SET OF Char;
    I         : Integer;

BEGIN
  Lowercase := ['a'..'z'];
  FOR I := 1 TO Length(Target) DO
    IF Target[I] IN Lowercase THEN
      Target[I] := Chr(Ord(Target[I]) - 32)
END;
```

FreePascal has a built-in procedure, **UpCase**, which will accomplish the same thing as the expression

```
Chr(Ord(Target[I]) - 32)
```

in the procedure above. However, the expression is a good example of the use of **Chr** and **Ord**, one right beside the other!

11.6. Pred & Succ

We discussed these two standard functions informally in Section 9.4, in connection with **FOR** loops. Now it's time for a closer look.

One of the properties of an ordinal type is that its values exist in a fixed and well-defined order. In other words, for type **Integer**, 3 comes after 2, not before. For type **Char**, 'Q' follows 'P' which follows 'O', and so on. The order is always the same.

This order is called the *collating sequence* or *collating order* of an ordinal type. Given a value of an ordinal type, **Ord** tells you which position that value occupies in its collating sequence. Given a value of an ordinal type, **Pred** and **Succ** return the next value before that value or after that value, respectively.

```
Pred('z')        { Returns 'Y' }
Succ('w')        { Returns 'x' }
```

```
Pred(43)        { Returns 42 }
Succ(19210)     { Returns 19211 }
Pred(Orange)    { Returns Red }
Succ(Green)     { Returns Blue }
Pred(Red)       { Undefined! }
```

This last example bears a closer look. The predecessor value of **Red** is undefined. Recall the definition of our enumerated type **Spectrum**:

```
Spectrum = (Red,Orange,Yellow,Green,Blue,Indigo,Violet);
```

Red is the very first value in the type. There is nothing before it, so **Pred(Red)** makes no sense in the context of type **Spectrum**. Similarly, **Succ(Violet)** makes no sense, since there is no value in **Spectrum** after **Violet**.

Pred(<value>) of the first value of an ordinal type is undefined. **Succ(<value>)** of the last value of an ordinal type is undefined.

Pred and **Succ** provide a means of "stepping through" an ordinal type to do some repetitive manipulation on a range of the values in that ordinal type. For example, in printing out the names on a telephone/address list, we might want to put a little header before the list of names beginning with 'A', and then before the list of names beginning with 'B', and so on. Assuming that the names are stored in sorted order in an array, we might work it this way:

```
VAR
  Names      : ARRAY[1..200] OF String[35];
  NameCount : Integer;



PROCEDURE Header(Ch : Char);

BEGIN
  Write(LST,Chr(13),Chr(10));
  Writeln(LST,'[',Ch,']-----------------------------')
END;



PROCEDURE PrintBook(NameCount : Integer);

VAR I     : Integer;
    Ch    : Char;
    AName : String[35];

BEGIN
  Ch := 'A';
  Header(Ch);
```

```
    FOR I := 1 TO NameCount DO
      BEGIN
        AName := Names[I];
        IF AName[1] <> CH THEN
          REPEAT
            CH := Succ(CH);
            Header(Ch)
          UNTIL AName[1] = CH;
        Writeln(LST,AName)
      END
END;
```

Assume that the array **Names** has been filled somehow with names, and that the number of names has been placed in **NameCount**. The names must be in sorted order, last name first. When **PrintBook** is invoked, the list of names in **Names** is printed on the system printer, with a header for each letter of the alphabet:

```
[A]-----------------------------
Albert*Eddie
Aldiss*Brian
Anthony*Piers

[B]-----------------------------
Brooks*Bobbie
Bentley*Mike

[C]-----------------------------
Chan*Charlie
Charles*Ray
Cabell*James Branch

[D]-----------------------------

[E]-----------------------------

[F]-----------------------------
Farmer*Philip Jose
Flor*Donna
```

and so on. Letters for which no names exist in the list will still have a printed header on the list. The printed listing, if cut into memo-book sized sheets, would make the core of a "little black book" for names and addresses.

Look at the listing of PrintBook. **Ch** is given an intial value of A. As the names are printed, the first letter of the each name is compared to the letter stored in **Ch**. If they don't match, the loop

```
REPEAT
```

```
  Ch := Succ(Ch);
  Header(Ch)
UNTIL AName[1] = Ch;
```

is executed. The letter in **Ch** is "stepped" along the alphabet until it "catches up" to the first letter in **AName**. For each step along the alphabet, a header is printed.

We might have written **Ch := Chr(Ord(Ch)+1)** instead of **Ch := Succ(Ch)**. **Succ** provides a much crisper notation. And because **Chr** does not work with enumerated types, **Succ** is the *only* way to step along the values of a programmer-defined enumerated type like **Spectrum**.

11.7. ODD

The last of the standard functions from ISO Standard Pascal that we'll talk about is a transfer function: **Odd(X)**. Here, **X** is an integer value. If **X** is an odd value **Odd(X)** returns the **Boolean** value **True**. If **X** is an even value, **Odd(X)** returns **False**.

Odd is thus a way of expressing an integer value as a value of type **Boolean**. Any even number can express the value **False**, and any odd number can express the value **True**. Although it doesn't fit the classic definition of "even" as "divisible by two," 0 is considered an even number by virtue of lying between two odd numbers, 1 and -1.

11.8. FRAC & INT

FreePascal includes a pair of functions originally introduced by Turbo Pascal: **Frac** and **Int**. With these two functions you can "take apart" a real number into its whole number part and its fractional part.

- • **Frac(R)** returns the fractional part of real number **R** . In other words, **Frac(24.44789)** would return 0.44789.
- • **Int(R)** returns the whole number part of real number **R**. In other words, **Int(241.003)** would return 241.0.

Both **Frac** and **Int** return values of type **Real**. You cannot directly assign the return values from either of these functions to an integer type. This seems counterintuitive for a function named **Int**; however, it's true: **Int** returns a real number type. For returning the whole number part of a real number to an integer variable, use the function **Trunc** instead.

11.9. PI

FreePascal includes a standard function **Pi** that returns the value of pi to as many significant figures as the variable to which it returns the value may express. The value returned is a real number type and so may not be assigned to any integer type, but (as with any real number type) may be assigned to the IEEE math coprocessor types **Single**, **Double**, **Extended** and **Comp**. **Pi** is one of the functions in the **Standard** unit.

You should keep in mind that differences in precision in the type receiving a value from Pi will affect the exact decimal values returned, although for all but the most exacting requirements these differences can be ignored.

To illustrate this issue, the following code displays the actual values returned by function **Pi** to the various real number types:

```
VAR
  RS : Single;
  RR : Real;
  RD : Double;
  RE : Extended;
  RC : Comp;

  RS := Pi; RR := Pi; RD := Pi; RE := Pi; RC := Pi;
  Writeln('Type Single:   ',RS:4:25);
  Writeln('Type Real:     ',RR:4:25);
  Writeln('Type Double:   ',RD:4:25);
  Writeln('Type Extended: ',RE:4:25);
  Writeln('Type Comp:     ',RC:4:25);
```

And here is the screen output you'll see:

```
Type Single:   3.141592741012573240
Type Real:     3.141592653588304530
Type Double:   3.141592653589793120
Type Extended: 3.141592653589793240
Type Comp:     3.000000000000000000
```

Note that type **Comp** can accept a value from Pi without triggering an error, but will not store the fractional part!

11.10. INC & DEC

Turbo Pascal 4.0 added two new standard procedures to its already considerable toolkit: **Inc** and **Dec**. They both operate only on ordinal types. Their function is simple: **Inc** increments an ordinal value, and **Dec** decrements an ordinal value.

Functionally, they are equivalent to **Pred** and **Succ** except that **Pred** and **Succ** are functions rather than procedures.

In other words, **Inc** and **Dec** may stand alone as statements:

```
VAR
  I : Integer;
  Ch : Char;

I := 17;
Ch := 'A';
Inc(I);      { I now contains 18 }
Dec(Ch);     { Ch now contains '@' }
```

Pred and **Succ**, by contrast, need to be placed on the right side of an assignment statement, either alone or inside of an expression:

```
I := 17;
Ch := 'A';
I := Succ(I);     { I now contains 18 }
Ch := Pred(Ch);   { Ch now contains '@' }
```

Why use **Inc** and **Dec** if the standard and much more portable **Pred** and **Succ** are available? Only one reason: speed. In most FreePascal platforms, **Inc** and **Dec** generate faster code than **Pred** and **Succ**. Why this is so lies in the details of compiler code generation, and so is beyond the scope of this book. In truth, on modern CPUs the difference won't become apparent unless you're making a lot more use of the functions than any ordinary program is ever likely to. Only if your code may need to be compiled with a different compiler will **Pred** and **Succ** have a distinct advantage.

11.11. RANDOM NUMBER FUNCTIONS

Built into FreePascal are two functions that return pseudorandom numbers, **Random** and **Random(I)**. **Random** returns a real number, and **Random(I)** returns an integer.

Random returns a pseudorandom number of type **Real** that is greater than or equal to zero and less than one. This statement:

```
FOR I := 1 TO 5 DO Writeln(Random);
```

might display:

```
7.0172090270E-01
```

```
7.3332131305E-01
8.0977424840E-01
6.7220290820E-01
9.2550002318E-01
```

The E-01 exponent makes all these numbers fall in the range 0.0 - 0.9999999999. All random numbers returned by the function **Random** fall within this range, but of course if you need random real numbers in another range, you need only shift the decimal point the required number of places to the right.

Random(I) returns random integers. The parameter is an integer that sets an upper bound for the random numbers returned by the function. **Random(I)** will return a number greater than or equal to zero and less than **I**. **I** may be any integer up to the maximum value that the integer type used for **I** can express. Although you can pass a negative number to **Random** without triggering an error, the returned value will always be the maximum value expressible in that integer type. There is no way to make **Random(I)** return negative random numbers.

There is frequently a need for a random number in a particular range, say between 15 and 50 or between 100 and 500. The procedure **Pull** shown below meets this need by extracting random integers until one falls in the range specified by **Low** and **High**.

```
{<<<< Pull >>>>}

{ From: FREEPASCAL FROM SQUARE ONE  }
{ by Jeff Duntemann -- Last mod 2/24/2018 }

FUNCTION Pull(Low,High : Integer) : Integer;

VAR
  I : Integer;

BEGIN
  REPEAT                      { Keep requesting random integers until }
    I := Random(High + 1); { one falls between Low and High }
  UNTIL I >= Low;
  Pull := I
END;
```

Randomize

The **Randomize** procedure exists because FreePascal's random numbers, like all random numbers generated in software, are not truly random but only *pseudorandom*, which means that a series of such numbers approximates randomness. The real issue is that a series of pseudorandom numbers may well repeat itself each time the program is run, unless the random number generator is "reseeded" with a new seed value. This is the job of **Randomize**.

Internally, what **Randomize** does is calculate a seed value from the computer's system clock, and place it in a predefined system variable called **RandSeed**. You don't have to access **RandSeed** at all. Each time you call **Randomize**, **RandSeed** gets a new seed value based on the current value in the system clock, which changes constantly.

Randomize should be called at least once in every program that makes use of random numbers, and to make your pseudorandom numbers more nearly random, might be called each time you want a new random number or series of random numbers.

A dice game

The following program shows one use of random numbers in a game situation. **Rollem** simulates the roll of one or more dice—up to as many as will fit across the screen. Procedure **Roll** may be placed in your function/procedure library and used in any game program that must roll dice in a visual manner. It will display a number of text-mode dice at location X,Y on the screen, where X,Y are the coordinates of the upper left corner of the first die. The **NumberOfDice** parameter tells **Roll** how many dice to roll; there is built-in protection against attempting to display more dice than space to the right of X will allow.

Rollem is also a good exercise in **REPEAT/UNTIL** loops. The **MakeBox** procedure was described in Section 10.2, as part of the **BoxTest** program..

```
{---------------------------------------------------------------}
{                         Rollem                                }
{                                                               }
{   A dice game to demonstrate random numbers and box draws     }
{                                                               }
{                         by Jeff Duntemann                     }
{                         FreePascal 3.0.4                      }
{                         Last update 2/24/2018                 }
{                                                               }
{   From: FREEPASCAL FROM SQUARE ONE  by Jeff Duntemann         }
{---------------------------------------------------------------}
```

```pascal
PROGRAM Rollem;

USES Crt,BoxStuff;

CONST
  DiceFaces : ARRAY[0..5,0..2] OF STRING[5] =
                   (('     ',' o ','     '),  { 1 }
                    ('o   ','     ','   o'),  { 2 }
                    ('    o',' o ','o    '),  { 3 }
                    ('o  o','     ','o  o'),  { 4 }
                    ('o  o',' o ','o  o'),  { 5 }
                    ('o o o','     ','o o o')); { 6 }


TYPE
  String80  = String[80];


VAR
  I          : Integer;
  Quit       : Boolean;
  Dice       : Integer;
  DiceX      : Integer;
  Ch         : Char;
  Banner     : String80;




PROCEDURE Roll(X,Y         : Integer;
               NumberOfDice : Integer);

VAR I,J,Throw,XOffset : Integer;

BEGIN
  IF (NumberOfDice * 9)+X >= 80 THEN        { Too many dice horizontally    }
    NumberOfDice := (80-X) DIV 9;           { will scramble the CRT display! }
  FOR I := 1 TO NumberOfDice DO
    BEGIN
      XOffset := (I-1)*9;                   { Nine space offset for each die }
      MakeBox(X+XOffset,Y,7,5,PCLineChars); { Draw a die }
      Throw := Random(6);                   { "Toss" it  }
      FOR J := 0 TO 2 DO                    { and fill it with dots }
        BEGIN
          GotoXY(X+1+XOffset,Y+1+J);
          Write(DiceFaces[Throw,J])
        END
    END
  END;
```

```
BEGIN
  Randomize;                          { Seed the pseudorandom number generator }
  ClrScr;                             { Clear the entire screen }
  Quit := False;                      { Initialize the quit flag }
  Banner := 'GONNA Roll THE BONES!';
  MakeBox(-1,1,Length(Banner)+4,3,PCLineChars);        { Draw Banner box }
  GotoXY((80-Length(Banner)) DIV 2,2); Write(Banner);  { Put Banner in it }
  REPEAT
    REPEAT
      FOR I := 6 TO 18 DO       { Clear the game portion of screen }
        BEGIN
          GotoXY(1,I);
          ClrEol
        END;
      GotoXY(1,6);
       Write('>>How many dice will we Roll this game? (1-5, or 0 to exit):
');
      Readln(Dice);
      IF Dice = 0 THEN Quit := True ELSE  { Zero dice sets Quit flag }
       IF (Dice < 1) OR (Dice > 5) THEN { Show error for dice out of range }
          BEGIN
            GotoXY(1,23);
            Write('>>The legal range is 1-5 Dice!')
          END
    UNTIL (Dice >= 0) AND (Dice <= 5);
    GotoXY(1,23); ClrEol;        { Get rid of any leftover error messages }
    IF NOT Quit THEN              { Play the game! }
      BEGIN
        DiceX := (80-(9*Dice)) DIV 2;   { Calculate centered X for dice }
        REPEAT
          GotoXY(1,16); ClrEol;
          Roll(DiceX,9,Dice);                         { Roll & draw dice }
          GotoXY(1,16); Write('>>Roll again? (Y/N): ');
          Readln(Ch);
        UNTIL NOT (Ch IN ['Y','y']);
        GotoXY(1,18); Write('>>Play another game? (Y/N): ');
        Readln(Ch);
        IF NOT (Ch IN ['Y','y']) THEN Quit := True
      END
  UNTIL Quit       { Quit flag set ends the game }
END.
```

CHAPTER 12.
STRING FUNCTIONS

The earliest versions of Pascal, including ISO Standard Pascal, had no clean and easy way to deal with text in a program. Virtually all Pascal compilers since then have implemented strings that go well beyond the crude Packed Array of Characters (PAOC) string type that was all Standard Pascal had to offer. The primary method of string representation in FreePascal goes all the way back to UCSD Pascal in 1978, and has evolved through the several versions of Turbo Pascal and Delphi.

We looked at FreePascal's string data types in some detail in Section 8.6. I'll only provide a quick recap here: A variable of type **STRING** is actually an array of characters with a counter attached at the beginning to keep tabs on how many characters have been loaded into the array. The original Turbo Pascal type **STRING** had a physical length of 255 characters. In FreePascal, there are two major string types: **ShortString** (which is basically Turbo Pascal's original **STRING** type) and **ANSIString**, which can be as long as 4,294,967,295 characters. **ANSIString** is the default string type. In other words, when you define a variable as type **STRING**, it will be created as an **ANSIString** variable. If you want to use the older **ShortString** type, you have to explicitly declare a variable as type **ShortString**.

The $H compiler directive changes the default: $H+ makes the default string type ANSIString, and $H- makes the default ShortString.

12.1. LENGTH

The length counter of a string variable is accessed with the predefined string function **Length**, which is defined this way:

```
FUNCTION Length(Target : STRING) : Integer;
```

Length returns the value of the length counter as type **Integer**, which indicates the logical length of string **Target** at any given time. Note that the length of an empty string is 0.

```
VAR Counter : Integer;

Counter := Length(MyText);
```

In the old days, many people read the length counter of a short string by examining element 0 of the string:

```
Counter := Ord(MyText[0]);
```

The transfer function **Ord** is necessary here, because a string is an array of characters, including the **ShortString** length counter. **Ord** converts the length counter from type **Char** to type **Integer**.

Don't do this! In the old days there was only one string type, and all strings treated character 0 as the length counter. FreePascal defaults to the **ANSIString** type, which is not as simple as **ShortString**. There is no "easy" way to read the length counter of an **ANSIString** by examining it. Use the **Length** function only.

The following procedure **CapsLock** accepts a string parameter and returns it with all lower-case letters changed to their corresponding upper-case letters. Note the use of the **Length** function:

```
PROCEDURE CapsLock(VAR MyString : OpenString);
VAR
  I : Integer;
BEGIN
  FOR I := 1 TO Length(MyString) DO
    MyString[I] := UpCase(MyString[I]);
END;
```

CapsLock makes use of a convenient built-in function, **UpCase**. UpCase accepts a character value as a parameter and returns the uppercase equivalent of that character, if the parameter is lowercase. If the parameter is *not* a lowercase character, it is returned unchanged.

Open string parameters were introduced with Borland Pascal 7.0 and are supported in FreePascal, via the predefined type **OpenString**. In Turbo Pascal, to pass strings physically shorter than 255 characters to **CapsLock** (or any subprogram with a string **VAR** parameter) you need to relax strict type checking with the {$V-} compiler command. If you're updating older Pascal code to FreePascal, keep this in mind.

12.2. STRING CONCATENATION

Concatenation is the process of taking two or more strings and combining them into a single string. FreePascal gives you two separate ways to perform this operation.

The easiest way to concatenate two or more strings is to use FreePascal's string concatenation operator (+). Many BASIC interpreters also use the plus symbol to concatenate strings. Simply place the string variables in order, separated by the string concatenation operator:

```
BigString := String1 + String2 + String3 + String4;
```

Variable **BigString** should, of course, be large enough to hold all the variables you intend to concatenate into it. If the total length of all the source strings is greater than the physical length of the destination string, all data that will not fit into the destination string is truncated off the end and ignored.

The built-in function **Concat** performs the same function as the string concatenation operator. It's considered a "legacy feature" because several older compilers, including UCSD Pascal, included it. Unless you're porting (very) old code to FreePascal, you're unlikely to need it, and the + operator is a great deal simpler and more intuitive. The following example shows how both string concatenation methods work:

```
VAR
  Subject,Predicate,Sentence : String[80];

Subject := 'Kevin the hacker';
Predicate := 'crashed the system';
Sentence  := Subject + Predicate;
Sentence := Concat(Sentence,', but brought it up again.');
Writeln(Sentence);
```

Here, two string variables and a string literal are concatenated into a single string variable. The output of the **Writeln** statement would be the single string "built" from the smaller strings via concatenation:

```
Kevin the hacker crashed the system, but brought it up again.
```

12.3. DELETE

Removing one or more characters from a string is the job done by the built-in **Delete** procedure, predefined this way:

```
PROCEDURE Delete(Target : STRING; Pos,Num : Integer);
```

Delete removes **Num** characters from the string **Target** beginning at the character number passed in **Pos**. The length counter of **Target** is updated to reflect the deleted characters.

```
VAR
  Magic : STRING;

Magic := 'Watch me make an elephant disappear...';
Delete(Magic,15,11);
Writeln(Magic);
```

Before the **Delete** operation, the string **Magic** has a length of 38 characters. When run, this example will display:

```
Watch me make  disappear...
```

The new length of **Magic** is set to 27.

One use of **Delete** is to remove "leading whitespace" from a string variable. Whitespace is a set of characters that includes space, tab, carriage return, and linefeed. Whitespace is used to format text files for readability by human beings. However, when that text file is read by a computer, the whitespace must be removed, as it tells the computer nothing.

The following procedure strips leading whitespace from a string variable:

```
PROCEDURE StripWhite(VAR Target : OpenString);
CONST
  Whitespace : SET OF Char = [#8,#10,#12,#13,' '];

BEGIN
  WHILE (Length(Target) > 0) AND (Target[1] IN Whitespace) DO
    Delete(Target,1,1)
END;
```

Whitespace is a set constant (see Section 8.7) containing the whitespace characters. **Delete(Target,1,1)** deletes one character from the beginning of string **Target**. The second character is then moved up to take its place. If it, too, is a whitespace character it is also deleted, and so on until a non-whitespace character becomes the first character in **Target**, or until **Target** is emptied of characters completely.

12.4. POS

Locating a substring within a larger string is handled by the built-in function **Pos**. **Pos** is predefined this way:

```
FUNCTION Pos(Pattern : <string or char>; Source : STRING) : Integer;
```

Pos returns an integer that is the location of the first occurrence of **Pattern** in **Source**. **Pattern** may be a string variable, a **Char** variable, or a string or character literal.

For example:

```
VAR
  ChX,ChY     : Char;
  Little,Big : STRING;

Big := 'I am an American, Chicago-born.  Chicago, that somber city.'
Little := 'Chicago';
ChX := 'g';
ChY := 'G';

Writeln('The position of ',Little,' in "Big" is ',Pos(Little,Big));
Writeln('The position of ',ChX,' in "Big" is ',Pos(ChX,Big));
Writeln('The position of ',ChY,' in "Big" is ',Pos(ChY,Big));
Writeln('The position of somber in "Big" is ',Pos('somber',Big));
Writeln('The position of r in "Big" is ',Pos('r',Big));
```

When executed, this example code will display the following:

```
The position of Chicago in "Big" is 19
The position of g in "Big" is 24
The position of G in "Big" is 0
The position of somber in "Big" is 48
The position of r in "Big" is 12
```

Pos *does* distinguish between upper and lower case letters. If **Pos** cannot locate **Pattern** in **Source**, it returns a value of 0.

12.5. COPY

Extracting a substring from within a string is accomplished with the **Copy** built-in function. **Copy** is predefined this way:

```
FUNCTION Copy(Source : STRING; Pos,Num : Integer) : STRING;
```

Copy returns a string that contains **Num** characters from **Source**, beginning at character **Pos** within **Source**:

```
VAR
  Roland,Tower : STRING;

Roland := 'Childe Roland to the Dark Tower came!';
Tower := Copy(Roland,22,10);
Writeln(Tower);
```

When run, this example will print:

```
Dark Tower
```

In this example, **Index** and **Size** are passed to **Copy** as constants. They can also be passed as integer variables or expressions. The following function accepts a string containing a file name, and returns a string value containing the file extension. (The extension is the part of a file name from the period to the end; in "sample.txt" the extension is ".txt".)

```
FUNCTION GetExt(FileName : STRING) : STRING;

VAR
  DotPos : Integer;

BEGIN
  DotPos := Pos('.',FileName);
  IF DotPos = 0 THEN GetExt := '' ELSE
    GetExt := Copy(FileName,DotPos,(Length(FileName)-DotPos)+1);
END;
```

GetExt first tests to see if there is, in fact, a period in **FileName** at all. (File extensions are optional.) If there is no period, there is no extension, and **GetExt** is assigned the null string. If a period is there, **Copy** is used to assign to **GetExt** all characters from the period to the end of the string.

Since the length of a file extension may be 2, 3, or 4 characters, the expression **(Length(FileName)-DotPos)+1** is needed to calculate just how long the extension is in each particular case. If **Index** plus **Size** is greater than the logical length of **Source**, **Copy** truncates the returned string value to whatever characters lie between Index and the end of the string.

12.6.INSERT

A string can be added to the end of another string by using the **Concat** function. Copying a string into the middle of another string (and not simply tacking it on at the end, as with concatenation) is done with the **Insert** procedure.

Insert is predefined this way:

```
PROCEDURE Insert(Source     : STRING;
                 VAR Target : STRING;
                 Position   : Integer);
```

When invoked, **Insert** copies **Source** into **Target** starting at position **Position** within **Target**. All characters in **Target** starting at position **Position** are moved forward to make room for the inserted string, and **Target**'s length counter is updated to reflect the addition of the inserted characters.

```
VAR
  Sentence,Ozzie : STRING;

Sentence := 'I am King of Kings.';
Ozzie := 'Ozymandias, ';
Insert(Ozzie,Sentence,6);
Writeln(Sentence);
```

The output from this example would be:

```
I am Ozymandias, King of Kings.
```

If inserting text into **Target** gives **Target** more characters than it can physically contain, **Target** is truncated to its maximum physical length. Characters that would fall beyond the physical lenth are lost.

Here's an example. The $H- compiler switch means that the string variables **Fickle** and **GOP** will be declared as short strings.

```
$H-

VAR
  Fickle,GOP : STRING[18];

Fickle := 'I am a Democrat.';
GOP := 'Republican.';
Insert(GOP,Fickle,8);
Writeln(Fickle);
```

This prints:

```
I am a Republican.
```

Note in this example that the string "Democrat." was not overwritten; it was pushed off the end of string **Fickle** into nothingness. After the insert, **Fickle** should have contained

```
I am a Republican.Democrat.
```

however, **Fickle**, defined as **STRING[18]**, is only 18 physical characters long. "I am a Republican." fills it completely. "Democrat." was lost to truncation.

12.7. STR

It is important to remember that a number and its string equivalent are not interchangeable. In other words, the integer 37 and its string representation, the two ASCII characters '3' and '7' look the same on your screen but are completely incompatible in all ways but that.

FreePascal provides a pair of procedures for translating numeric values into their string equivalents, and vise versa. Translating a numeric value to its string equivalent is done with the procedure **Str**. **Str** is predefined this way:

```
PROCEDURE Str(<formatted numeric value>; VAR ST : STRING);
```

The formatted numeric value can be either an integer or a real number. It is given as a write parameter. (See Section X.X for a complete discussion of write parameters as they apply to all simple data types, numeric and non-numeric.) Briefly, a write parameter is an expression that gives a value and a format to express it in. The write parameter **I:7** (assuming **I** was previously declared an integer) right-justifies the value of **I** in a field seven characters wide. **R:9:3** (assuming **R** was declared **Real** previously) right-justifies the value of **R** in a field 9 characters wide with three figures to the right of the decimal place.

The use of **Str** is best shown by a few examples:

```
CONST
  Bar = '|';

VAR
  R   : Real;
  I   : Integer;
  TX  : STRING[30];

R := 45612.338;
I := 21244;

Str(I:8,TX);
Writeln(Bar,TX,Bar);     { Displays:  |   21244| }

Str(I:3,TX);
Writeln(Bar,TX,Bar);     { Displays:  |21244|  }

Str(R,TX);
Writeln(Bar,TX,Bar);     { Displays:  |   4.5612338000E+04| }

Str(R:13:4,TX);
Writeln(Bar,TX,Bar);     { Displays:  |    45612.3380| }
```

Note from the third example that if you do not specify any format for a real number, the default format will be scientific notation in a field eighteen characters wide.

12.8. Val

Going in the other direction, from string representation to numeric value, is accomplished by the **Val** procedure. **Val** is predeclared this way:

```
PROCEDURE Val(ST : STRING; VAR <numeric variable>; VAR Code : Integer);
```

Val's task is somewhat more complicated than **Str**'s. For every numeric value there is a string representation that may be constructed from it. The reverse is not true; there are many string constructions that cannot be evaluated as numbers. So **Val** must have a means of returning an error code to signal an input string that cannot be evaluated as a number. This is the purpose of the **Code** parameter.

   If the string is evaluated without any problem, **Code**'s value is 0 and the numeric equivalent of the string is returned in the numeric variable. If FreePascal finds that it cannot evaluate the string to a number, **Code** returns the character position of the first character that violates the evaluation scheme. The numeric variable in that case is undefined:

```
PROGRAM Evaluator;

VAR
  SST    : STRING;
  R      : Real;
  Result : Integer;

BEGIN
  REPEAT
    Write('>>Enter a number in string form: ');
    Readln(SST);
    IF Length(SST) > 0 THEN
      BEGIN
        Val(SST,R,Result);
        IF Result <> 0 THEN
          Writeln
          ('>>Cannot evaluate that string.  Check character #',Result)
        ELSE
          Writeln
          ('>>The numeric equivalent of that string is ',R:18:10)
      END
  UNTIL Length(SST) = 0
END.
```

   This little program will allow you to experiment with **Val** and see what it will accept and what it will reject. One unfortunate shortcoming of **Val** is that it considers commas an error. A string like '5,462,445.3' will generate an error on character #2.

Table 12.1 contains a summary of FreePascal's built-in string-handling routines.

Table 12.1. Built-in string-handling routines

```
FUNCTION Concat(Source1,Source2...SourceN : STRING) : STRING

FUNCTION Copy(Source : STRING; Index,Size : Integer) : STRING

PROCEDURE Delete(Target : STRING; Index,Size : Integer)

PROCEDURE Insert(Source     : STRING;
                 VAR Target : STRING;
                 Index      : Integer)

FUNCTION Length(Source : STRING) : Integer

FUNCTION Pos(Pattern : STRING or Char;
             Source  : STRING) : Integer

PROCEDURE Str(Num : <write parameter>, VAR StrEquiv : STRING)

PROCEDURE Val(Source       : STRING;
              VAR NumEquiv : <Integer or Real>;
              VAR Code     : Integer)
```

12.9. MORE EXAMPLES OF STRING MANIPULATION

Perhaps the first ambitious program most beginning programmers attempt is a name/address/phone number manager. Sooner or later, in designing such a program, the problem comes up: How to sort the list on the name field, when names are stored first name first and sorted last name first?

Storing the first name in a separate field is no answer—suppose you want to store The First National Bank of East Rochester? What is its first name?

The best solution I have found is to store the name last name first, with an asterisk (*) separating the last and first names. For example, Jeff Duntemann would be stored as Duntemann*Jeff. Clive Staples Lewis would be stored as Lewis*Clive Staples. Names maintained in this order are easily sorted by last name. All we need is a routine to turn the inside-out name rightside-in again.

The following routine does just that—and demonstrates **Pos**, **Copy**, **Delete**, and **Concat**, all in four lines!

```
{<<<< RvrsName >>>>}
{ From: FreePascal From Square One  }
{ by Jeff Duntemann -- Last mod 6/10/2018 }

PROCEDURE RvrsName(VAR Name : OpenString);

VAR
  TName : String;

BEGIN
  IF Pos('*',Name) <> 0 THEN
    BEGIN
      TName := Copy(Name,1,(Pos('*',Name)-1));
      Delete(Name,1,Pos('*',Name));
      Name := Concat(Name,' ',TName)
    END
END;
```

The theory is simple: If there is no asterisk in the name, it's something like "Granny Maria's Pizza Palace" and needs no reversal. Hence the first test. If an asterisk is found, the last name up to (but not including) the asterisk is copied from **Name** into **TName**, a temporary string. Then the last name is deleted from **Name**, up to and including the asterisk. What remains in **Name** is thus the first name. Finally, concatenate **TName** (containing the last name) to **Name** with a space to separate them. The name is now in its proper, first-name-first form.

Obviously, if you try to store and sort on a name of some sort that rightfully contains an asterisk, the name is going to be mangled by **RvrsName**.

A case adjuster function for strings

FreePascal provides a built-in character function called **UpCase**, predeclared this way:

```
FUNCTION UpCase(Ch : Char) : Char;
```

UpCase accepts a character **Ch** and returns its upper-case equivalent as the function return value. If **Ch** is already upper-case, or a character with no upper-case equivalent, (numerals, symbols, and so on) the character is returned unchanged.

UpCase is a character function, but it suggests that a string function could be built that accepts an arbitrary string value and returns that value converted to upper case. And although no "down-case" function exists in FreePascal, an equivalent is not hard to put together. A two-way case adjuster function looks like this:

```
{<<<< ForceCase >>>>}
{  From: FreePascal From Square One  }
{  by Jeff Duntemann -- Last mod 6/10/2018 }

FUNCTION ForceCase(Up : BOOLEAN; Target : STRING) : STRING;

CONST
  Uppercase : SET OF Char = ['A'..'Z'];
  Lowercase : SET OF Char = ['a'..'z'];

VAR
  I : INTEGER;

BEGIN
  IF Up THEN FOR I := 1 TO Length(Target) DO
    IF Target[I] IN Lowercase THEN
      Target[I] := UpCase(Target[I])
    ELSE { NULL }
  ELSE FOR I := 1 TO Length(Target) DO
    IF Target[I] IN Uppercase THEN
      Target[I] := Chr(Ord(Target[I])+32);
  ForceCase := Target
END;
```

In FreePascal, functions may return string values the same as any other values. However, the string type must either be the default type **STRING** or have been declared before the declaration of your string function. In other words, if you wish your function to return a string with a physical length of 80 you must have declared a string type with that physical length:

```
TYPE
  String80 = STRING[80];
```

You cannot use the bracketed string-length notation on a string function return value. That is, you could not have declared **ForceCase** this way:

```
FUNCTION ForceCase(Up     : Boolean;
                   Target : String80) :  String[80];
                                             {^Invalid!}
```

ForceCase will convert all uppercase characters in a string to lowercase, or all lowercase characters in a string to uppercase, depending on the **Boolean** value of parameter **Up**. If **Up** is true, lower case is forced to uppercase. Otherwise, uppercase is forced to lowercase. The string **Target** is scanned from character 1 to its last character, and any necessary conversion of character case is done character-by character. The "down-case" function is done by taking advantage of the ordering of the ASCII character set, in that lowercase characters have an ASCII value 32 higher

than their uppercase counterparts. Add 32 to the ordinal value of an uppercase character, and you have the ordinal value of its lowercase equivalent.

Also note that although the parameter string **Target** is modified during the scan, the modifications are not made to the actual parameter itself, since **Target** was passed by value, not by reference. **ForceCase** received its own private copy of **Target**, which it could safely change without altering the "real" **Target**. See Section 10.3 for more on the passing of parameters by value or by reference.

Accessing command-line strings

Most operating systems allow some sort of program access to the command line tail; that is, the optional text that may be typed after the program name when invoking a program from the operating system command prompt:

```
CASE DOWN B:FOOFILE.TXT
```

In this example, the characters typed after the program name "CASE" constitute the command line tail:

```
DOWN B:FOOFILE.TXT
```

FreePascal provides a very convenient method of getting access to the command line tail. Two predefined functions are connected with the command line tail: ParamCount and ParamStr. They are predeclared this way:

```
FUNCTION ParamCount : Integer;
FUNCTION ParamStr(ParameterNumber : Integer) : STRING;
```

The function **ParamCount** returns the number of parameters typed after the command on the operating system command line. Parameters must have been separated by spaces or tab characters to be considered separate parameters. Commas, slashes, and other symbols on will not delimit separate parameters!

ParamStr returns a string value that is one of the parameters. The number of the parameter is specified by **ParameterNumber**, starting from 1. If you typed several parameters on the command line, for example:

```
ParamStr(2)
```

will return the second parameter. Note that **ParamStr[0]** is the name of the program executed with the parameters starting at **ParamStr[1]**.

Keep this in mind: Always read the command line tail before opening your first disk file! The same area use to store the tail is also used in buffering disk accesses in some cases using some operating systems. The best way to avoid this sort of

trouble is to keep an array of strings large enough to hold the maximum number of
parameters your program needs, and read the parameters into the array as soon as
your program begins running. This is easy enough to do:

```
VAR
  I : Integer;
  ParmArray : ARRAY[1..8] OF STRING[80];

FOR I := 1 TO ParamCount DO
  ParmArray[I] := ParamStr(I);
```

Now you have the parameters safely in **ParmArray** and can examine and use them
at your leisure.

Don't Become Too Fond of Text Mode…

To keep this book shorter than *Borland Pascal 7 From Square One,* I cut out quite a bit
of material, including about half of this chapter. Specifically, I cut out some longish
code examples showing how to use Pascal's string functions to create text-mode
edit fields and data-entry screens. Text mode isn't used much anymore, especially
for data-entry screens. Once you begin writing GUI applications for Windows or
Linux, you'll find that Lazarus has a palette full of components providing data entry
fields and even whole text editors. In my next book, *Lazarus GUI Programming From
Square One,* I'll show how data entry screens and dialogs may be assembled from the
components included with the Lazarus product.

CHAPTER 13.
LOCALITY AND SCOPE

At the heart of structured programming is that old saw about the artful hiding of details. You want to be able to focus in on the level of detail where you're currently working, and not be excessively concerned with either the details down lower, or the larger view from above. On a purely structural level, the best way of hiding details is to divide a program into *subprograms* (that is, procedures and functions) and by grouping data into data structures like arrays, records, and (once you've had some experience) objects. When you need to think of the task that a subprogram does, you simply think of it as a little black box that does one or two well-defined things. You don't worry about what's inside the box—*unless* you're actually going to tinker with what's inside the box. That's when you open the box and take a look.

This sounds simple enough on the surface, but there are some subtle issues surrounding it that I found *very* confusing when I was first learning the Pascal language, back in the late 1970's. There's not a lot to discuss (which is why this chapter is so short) but what there is happens to be extremely important, especially if you expect to become a truly world-class programmer.

13.1. THE INNARDS OF A SUBPROGRAM

Functions and procedures aren't called *subprograms* for nothing. Their structure is almost identical to that of your Pascal main program: They have a name, they have definitions, and they have a body consisting of a compound statement bounded by reserved words **BEGIN** and **END**. They can even have their own subprograms, nested inside them like Chinese boxes.

Here's a subprogram we've seen before. I removed the comment header to save space:

```
PROCEDURE ShellSort(VAR SortBuf : KeyArray; Recs : Integer);

VAR
  I,J,K,L : Integer;
  Spread  : Integer;


PROCEDURE KeySwap(VAR RR,SS : KeyRec);

VAR
  T : KeyRec;

BEGIN
  T  := RR;
  RR := SS;
  SS := T
END;



BEGIN
  Spread := Recs DIV 2;    { First Spread is half record count }
  WHILE Spread > 0 DO      { Do until Spread goes to zero:      }
    BEGIN
      FOR I := Spread + 1 TO Recs DO
        BEGIN
          J := I - Spread;
          WHILE J > 0 DO
            BEGIN               { Test & swap across the array }
              L := J + Spread;
              IF SortBuf[J].KeyData <= SortBuf[L].KeyData THEN
                J := 0 ELSE
                KeySwap(SortBuf[J],SortBuf[L]);
              J := J - Spread
            END
        END;
      Spread := Spread DIV 2    { Halve Spread for next pass }
    END
END;
```

The **ShellSort** procedure defines five of its own variables, and has a subprogram of its own, **KeySwap**. The **KeySwap** procedure, moreover, defines its own variable, **T**. The **KeySwap** procedure is only called from one place inside **ShellSort**. Its whole job is to hide the details of swapping two keys so that those details don't get in the way while you're reading **ShellSort**. At the time the swap happens, precisely how the swap happens is unimportant. You simply need to know that the two parameters are exchanged. The **ShellSort** procedure itself exists to hide the

details of sorting an array of keys. When you're writing a data manager program that keeps a data file and a sorted key file, you don't necessarily want to be bothered with the details of how the sort happens. That's why you only need to see this much of **ShellSort** when you actually want to use it to perform a sort:

```
ShellSort(MySortBuf,KeyCount);
```

All you need to know at this point is that you're going to sort the key array **MySortBuf**, which contains **KeyCount** key records. The details of how the sort happens are irrelevant. Later on, if you want to tinker with the sort routine a little bit, you can go to the source code file that contains the **ShellSort** procedure and work on it. But when you're simply building a sort call into a program you're writing, the call itself is all you need.

13.2. GLOBAL VS. LOCAL

When one subprogram is defined inside another, we say that the inner subprogram is *local* to the outer one. The same term applies to variables and other definitions that exist inside a subprogram. The variables **I**,**J**,**K**,**L**, and **Spread** are local to **ShellSort**. Variable **T** is local to **KeySwap**.

A different term is applied to definitions that you place in the main program itself. These are called *global* definitions. The difference between local and global follows the sense of the terms themselves: Global definitions are "known" (we say, *visible*) throughout the entire program and everything within it. Local definitions are visible only from inside their containing entity.

The term "visible" is a technical one here, and it amounts to an ability to make a reference to an identifier; that is, to read it or write a new value to it. If you can reference an identifier from some place in a program, the identifier is visible from that place in the program.

The local variable **T** defined by **KeySwap** is an excellent example. **KeySwap** uses **T** as a temporary bucket to hold a key record during the swap process. Inside **KeySwap**, **T** is obviously visible, because **KeySwap's** internal logic uses it.

Now, how about from inside **ShellSort**? Could a statement in **ShellSort**'s body reference **T**? No! **T** is local to the **KeySwap** procedure, and is only visible from within **KeySwap**. If you tried to read from or write to **T** from **ShellSort**'s body, the compiler would issue an unknown identifier error. And that's the truth: From within **ShellSort**'s procedure body, variable **T** really is unknown. It's local to **KeySwap**, and unknown anywhere outside **KeySwap**'s procedure body. The same would be true from the main program block: You could not reference **T** at all.

Scope

This "visibility" property of a Pascal identifier is called its *scope*. The scope of an identifier is that area of the program from which the identifier can be referenced. As we saw in the last section, the scope of **T** is limited to the **KeySwap** procedure alone. The scope of the **KeySwap** procedure is limited to **ShellSort**. The main program cannot directly call **KeySwap**. (Nor could some other procedure, like **QuickSort**, call **KeySwap**.) Only **ShellSort** can call **KeySwap**.

In general (and more technical) terms, the scope of an identifier is limited to the block in which it is defined, and to all blocks defined inside that block.

What this means is that scope extends "down" the nesting hierarchy, but not "up." That is, the scope of variable **Spread** in **ShellSort** extends down into **KeySwap**, but the scope of **T** in **KeySwap** does not extend up into **ShellSort**. If it needed to, **KeySwap** could read the current value in **Spread**, but nothing in **ShellSort** could read the current value of **T**.

13.3. Identifier conflicts and scope

There's an interesting consequence of Pascal's scoping rules: You can have more than one identifier in a given program with the exactly the same name, and nobody will complain. *You* might complain, if you don't understand the rules—which is the purpose of this chapter!

Let's take a look at a short and highly contrived Pascal program to try and get a more precise handle on this:

```pascal
PROGRAM Hollow;

VAR
  Z     : Integer;
  Ch, Q : Char;
  Gonk  : String[80];


PROCEDURE LITTLE1;

VAR
  Z : Integer;

BEGIN
END;
```

```
PROCEDURE LITTLE2;

VAR
  Z : Integer;
  Q : Char;

BEGIN
END;



BEGIN     { Main for Hollow }
  Little1;
  Writeln('>>We are the hollow programs,');
  Little2;
  Writeln('>>We are the stuffed programs.')
END.
```

Program **Hollow** is nothing more than the merest skeleton of a program, constructed solely to illustrate the concept of identifier scope. If you're sharp and have looked closely at **Hollow**, you may be objecting to the fact that there are three instances of a variable named "**Z**"—and I already told you that Pascal does not tolerate duplicate identifiers. Well, due to Pascal's scoping rules, the three **Z**'s are not in fact duplicates at all.

Up near the top of the source code file, in program **Hollow**'s own variable list, is a variable named **Z**. In **Little1** is a variable named **Z**, but *this* **Z** is local to **Little1**. **Little2** also has an **Z** that is local to **Little2**. Each **Z** is "known" only in its own neighborhood, and the extent of that neighborhood is its scope. The scope of **Little1**'s **Z** is *only* within **Little1**. Likewise, **Little2**'s **Z** is known only within **Little2**. You cannot access the value of **Little2**'s **Z** (or, for that matter, *any* variable declared within **Little2**) from within **Little1**. Furthermore, while you're in the main program, you cannot access either of the two **Z**'s that are local to **Little1** and **Little2**. They might as well not exist until you enter one of the two procedures in which those local **Z**'s are declared. (In fact, in reality they *don't*. Stay tuned.)

This get a little slipperier when you consider the **Z** belonging to program **Hollow** itself. That **Z**'s neighborhood encompasses the entire program, which includes both **Little1** and **Little2**. So while we're within **Little1** or **Little2**, which **Z** is the *real* **Z**? Plainly, we need a rule here, and the rule is called *precedence*. It's just this: *When the scopes of two identical identifiers overlap, the most local identifier takes precedence*. In other words, while you're within **Little1**, **Little1**'s own local **Z** is the only **Z** you can "see." The **Z** belonging to **Hollow** is hidden from you while you're within the scope of a more local **Z**. **Hollow**'s **Z** doesn't go away and doesn't change; you simply can't look at it nor change it while **Little1**'s **Z** takes precedence.

The Mike Smith Metaphor

Think of it this way: There are way too many Mike Smiths in the world. A large national liquor company based in New York City has a vice president named Mike Smith. The company also has two regional salesmen named Mike Smith, one in Chicago and one in Geneseo, New York. When you're at corporate headquarters in New York and mention Mike Smith, everyone assumes you mean the Vice President of Whisky Keg Procurement. However, if you're in Chicago and mention Mike Smith to a liquor store owner, he thinks you mean the skinny chap who sells him Rasputin Vodka. He's probably never heard of the VP or the salesman in Geneseo. Furthermore, if you're in Geneseo and mention Mike Smith, the restaurant owners think of the fellow with the red beard who distributes Old Tank Car wines. They don't know (and could not care less about) the VP of Whisky Keg Procurement or the vodka salesman in Chicago.

Look back at **Hollow** for a moment and consider the variable **Q**. **Hollow** has a **Q**. **Little2** also has one. **Little1** does not. Within **Little2**, **Little2**'s **Q** is king, and **Hollow**'s **Q** is hidden away. However, **Little1** can read and change **Hollow**'s **Q**. There is no precedence conflict here because there is no **Q** in **Little1**. Since the scope of **Hollow**'s **Q** is the entire program, any function or procedure within **Hollow** can access **Hollow**'s **Q** as long as there is no conflict of precedence. We say that **Hollow**'s **Q** is global to the entire program. In the absence of precedence conflicts, **Hollow**'s **Q** is "known" throughout **Hollow**.

Hollow has a variable named **Gonk** that can be accessed from either **Little1** or **Little2** because it is the only **Gonk** anywhere within **Hollow**. With **Gonk**, the question of precedence does not arise at all.

Unless you can't possibly avoid it, don't make your procedures and functions read or change global variables (like **Ch** or **Gonk**) *unless* those variables are passed to the procedure or function through the parameter line. This prevents data "sneak paths" among your main program and procedures and functions. Such sneak paths are easy to forget when you modify a program, and may mess up legitimate changes to the program in (apparently) inexplicable ways.

To sum up:
- An identifier is local to the block (procedure, function, module, or program) in which it is defined. This block is the scope of that identifier.
- You cannot access a local identifier unless you are within that block, or contain that block.
- Where a duplicate identifier conflict of scope exists, the more local of the two is the identifier that you can access.

Why?

This can be a lot of abstract logic and rules to swallow without some firm peg in the real world to hang it on. I don't know about you, but understanding the physical reality of a program helps me understand its more abstract logic. So if you're feeling ambitious, I'll spend a few words explaining why Pascal does things this way.

Identifiers defined within a subprogram are not visible from outside that subprogram because until that subprogram is called, *its identifiers literally do not physically exist*. We who see the whole program and all of its subprograms laid out on the screen or on sheets of paper, all at once, sometimes forget the time-sequential nature of a Pascal program.

Global variables are allocated in an area of memory called the *data segment*. They are brought into existence when the program is loaded into memory, and remain in existence until the program hands control back to the operating system, or to the IDE. (The IDE can sometimes play games with a program such that the program and the IDE can bounce back and forth during the debugging process, and the IDE can "see" variables within the program. This is a separate issue, and involves some magic you won't be using in ordinary Pascal programming.) Local variables, by contrast, do not exist until the very moment that their subprograms are called.

When a subprogram is called, it receives a little slice of an area of memory called the *stack*. Inside that little slice of the stack are allocated its local variables. Now, the stack is strictly temporary, reusable storage, and when a subprogram finishes executing and returns control to its caller, the region of the stack that it had used is freed up for some other subprogram to use. *A subprogram's local variables exist on the stack only during the time that the subprogram is executing.* When the subprogram returns, its variables go *poof!* and are no longer anywhere to be seen.

We say that subprograms have a limited *lifetime*—the time that they are executing and own their little slice of the stack containing their local variables. Before and after that lifetime, a subprogram's local variables simply do not exist.

So Pascal's scoping rules are not simply the compiler being authoritarian. It can't allow you to reference something that doesn't exist yet or no longer exists. *Rules are for reasons!* Don't gripe about the rules. Strive to know the reasons.

Chapter 14.
FreePascal File I/O

A computer program is a universe unto itself. Within its bounds data structures are created, changed, and deleted. Calculations are performed and the results used in still more calculations. Lots goes on in a computer program's universe in the cause of getting a job done.

But it's out here in *our* universe where the job comes from and the finished work is needed. Somehow, some of the things living in a computer program have to cross the border between the program-universe and the real universe. The pathways between the computer program and the outside world are collectively called "input/output" or, more tersely, "I/O." In Standard Pascal, all I/O is handled under the umbrella category of *files*.

In this chapter I will explain "classic" Pascal file I/O, which (mostly) goes back to the dawn of Pascal time. I call it classic because newer file mechanisms were introduced with object-oriented programming (OOP), most of them in the form of *streams*. I don't have room to cover OOP in this book, so I can't cover streams yet. In my next book, which will cover OOP, software components, and GUI app creation in Lazarus via FreePascal, I'll take up the topic at length. Classic file I/O is a good way to learn about the concept of files, and might still be useful in small utilities.

14.1. The notion of a file

To most people, the word "file" conjures images of filing cabinets and various other places to store things away for future perusal and re-use. However, another connotation of the word "file" strikes closer to heart of the Pascal file: a long row of people or objects in a straight line, often in the process of moving from one place to another in an orderly fashion. Soldiers file from the parade ground to the mess hall, and so on.

A Pascal file is a stream of data, either coming or going. Not all "files" are data stored on disk. Your keyboard is a file: a stream of characters triggered by your fingers and

sent inward to the waiting program. Your CRT screen is a file: a flat field painted with visible symbols by a stream of characters sent from the program inside the computer. Your printer is a file: a device that accepts a stream of characters from the program and places them somehow onto a piece of paper.

The last example contains an important word: "device." Your keyboard is a device, as is your CRT screen, as is your printer, as is your modem. All are files that have one endpoint in the computer program and another endpoint in a physical gadget that absorbs or emits data. (Sometimes a file may do both.) This kind of file we call a "device file."

The other kind of file is the more familiar kind: a collection of data recorded as 1's and 0's on rotating magnetic disks or (increasingly) Flash memory devices like thumb drives, SD cards, and solid-state disks (SSDs). These are all colloquially called "disk files," even the ones not stored on any kind of disk. Classic files in FreePascal are of one type or another, either device files or disk files.

The three types of FreePascal file I/O

Like everything else in computer technology, the Pascal language has evolved hugely over time. It's been around since 1970, which (at this writing; 2025) is over fifty years ago. Modern compilers like FreePascal would be almost unrecognizable to Pascal's early adopters. The biggest single change in Pascal during its lifetime was the addition of object-oriented programming (OOP) back in the late 1980s. As I said earlier, OOP is such a big topic that it will have to wait for another book. The second biggest change has been in the ways that Pascal handles data files. There are three general categories of file management techniques in FreePascal: simple file I/O, stream file I/O, and database management.

- **Simple file I/O** dates back to the original design of the language. Pascal's file I/O in the 1970s was very simple and very crude. It depended on two routines, Get and Put, that are no longer present in modern compilers like FreePascal. In 1983, Borland's Turbo Pascal abandoned **Get** and **Put** and rolled simple file I/O into the **Read**, **Write**, **Readln**, and **Writeln** routines.

- **Stream file I/O** is an element of object-oriented programming, and was pioneered by Turbo Pascal 5.5 in 1989. Stream file I/O is implemented by **TStream** and its descendent types **TMemoryStream** and **TFileStream**. Because it makes heavy use of OOP techniques, I can't take it up in this book.

- **Database management** moves file I/O to a still higher level, where data is organized in rows and columns and managed by a *database engine*, which is generally (but not always) a separate program. Database management is not

inherently an OOP technology, but it is deep enough to demand an entire book to do it justice, even for beginners.

I say all this to advise caution: Don't spend a great deal of time exploring Pascal's simple file I/O if you intend to immediately go on to OOP-based programming or database work using Lazarus and its suite of software components. This book is about Pascal basics, and that includes simple file I/O. Be aware of it, understand how it works…but don't be content with simple file I/O when there's so much better to be had using OOP and database engines.

14.2. LOGICAL AND PHYSICAL FILES

ISO Standard Pascal does not make a distinction between device and disk files. In ISO Pascal, a file is a file—a stream of data either coming or going. One end of the line is inside the Pascal program, and Pascal simply doesn't care what sort of thing is on the other end of the line.

That's ISO. FreePascal has had to modify this notion a little, but the "everything Out There is a file" bias remains. By not getting involved in the deeper details of how a character is printed or sent to the screen, Pascal the language (that is, its workaday syntax) can present a uniform front to many different operating systems. This makes for less work in translating a Pascal program from one operating system platform to another.

This separation between *what* a file does and *how* it does it is sharply drawn in Pascal, between the concepts of "logical" and "physical" files. A *logical file* is a Pascal abstraction, beginning with the declaration of a file variable in the variable declaration part of a program:

```
StatData : FILE OF Integer;
```

This tells us that **StatData** is a file and contains integers, period. It says nothing about whether or not the file exists on a disk (or on some other type of storage, like an SSD or thumb drive), which disk, or how much data exists in the file.

That information falls under the realm of physical files. A *physical file* is an actual device (for device files) or an actual collection of data on some sort of physical storage medium. A physical disk file has a file name, a size, a record length, an access mode, (read only, read/write, hidden, system, etc.) perhaps a timestamp for last access and an interleave factor, and other things as well. It is *not* an abstraction, and by not being an abstraction a physical file must pay attention to all those gritty little details that make storing and retrieving data on disk possible.

All these little details are very much hardware- and operating-system dependent.

If Pascal had to take care of such details, it would be nearly an entirely separate language for each computer/operating system combo it ran on, as was very nearly the case with BASIC in its early days. So Pascal provides logical files, and the operating system provides physical files. When a logical file becomes associated with (or *assigned to*, as is often said) a physical file, a path is created between the program's inner universe and the outside world. That file is then said to be open.

14.3. DECLARING, ASSIGNING, AND OPENING FILES

Declaring a file gives the file a name and a data type:

```
FileVariableName : FILE OF <type>;
```

<type> may be any valid data type except a file type. (A file of files makes no logical sense and is illegal.) <Type> can be a standard type like **Integer** or **Boolean**, or it can be a type that you have defined, like records, sets, enumerated types or arrays. A file can be a file of only one type, however; you cannot declare a file of **Integer** and then write records or sets to it.

Declaring a file also creates a file buffer in your data area. This file buffer is a variable of the type that the file is declared to be. It is a "window" into the file; the actual open end of the data pipe between the program and the outside world.

Connecting logical to physical with Assign

Before you can open a file, there must first be a connection between a logical file that is declared in the program, and a physical file that exists outside the program. Creating this connection is the job of the built-in procedure Assign.

Assign takes two parameters: a file variable that has been declared in the program, and a string variable or literal that names a physical file:

```
PROCEDURE Assign(AFile : <filetype>; FileName : String);
```

Assign creates the File Information Block (usually called a FIB) which each file must have to be used by the program. Once associated with a physical file, a logical file may be opened for write access or read access. This is done with **Reset** and **Rewrite**.

Opening files with Reset and Rewrite

Assign does only half the job of opening a file; its job is limited to linking a logical file with a physical file. Getting a logical file ready to be written to or read from involves two built-in procedures: **Reset** and **Rewrite**.

An open file has in its FIB what we call a *file pointer*. This is not a Pascal pointer variable (I will not be covering pointers in this book) but rather a logical marker indicating which element in the file will be read or written to next.

Reset opens a file for reading. Some device files allow both read access and write access at the same time; these files are opened with Reset also.

Reset does not disturb the previous contents of a disk file. The file pointer is positioned at the first data item in the file. If the file is empty when Reset is executed, the Boolean function **EOF** (end of file) returns **True** for that file.

Some examples:

```
Assign(MyFile,'C:ADDRESS.TXT');
Reset(MyFile);
Assign(ConsoleInput,'CON');
Reset(ConsoleInput);
```

Reset can also be used on a file that is already open. For an open disk file, this will reposition the file pointer to the first record in the file. Performing a **Reset** on an open device file does nothing. If the device file is one of FreePascal's predefined logical devices, **Reset** will generate an error.

Rewrite opens files that are *only* to be written to, and not read from while they are currently open. When **Rewrite** is executed on a disk file, all previous contents of the disk file are overwritten and lost. Note that you may use either **Reset** or **Rewrite** on device files; the action on the device file is identical for both.

```
Assign(MyFile,'C:GRADES.DAT');    { Old grades are lost! }
Rewrite(MyFile);
Assign(BitBucket,'NUL');       { Anything written to BitBucket }
Rewrite(BitBucket);            { will quietly vanish away...   }
```

Input and Output

All Pascal programs are capable of accepting input from the keyboard and writing output to the screen. Two standard device files are always open for this purpose: **Input** and **Output**.

In some (generally old) Pascal compilers you must explicitly name **Input** and **Output** when using them with **Read**, **Write**, **Readln**, and **Writeln**:

```
Writeln(Output,'Files do it sequentially...');
```

In FreePascal, **Input** and **Output** are the default files for use with **Read**, **Write**, **Readln**, and **Writeln**. If you omit the name of a file in one of those statements, the

compiler assumes **Input** for **Read** and **Readln**, and **Output** for **Write** and **Writeln**. You need only write:

```
Writeln('Files do it sequentially...');
```

and FreePascal will know to send the quoted text line to your CRT screen.

Similarly, to accept input from the **Input** logical file, which is always connected to your keyboard and always open, you need only write:

```
Readln(AString);
```

and the program will wait while you type text at the keyboard, continuing on only when it detects that the Enter key was pressed, or when you have filled the string variable out to its maximum physical length. The text you typed will be immediately available in the variable **AString**. There is no need to include the identifier **Input** in the **Readln** statement, although you may if you wish—nothing will change.

Input and **Output** are considered text files, which means that only printable ASCII characters and a few control characters may be read from them or written to them. *However*, **Read**, **Readln**, **Write**, and **Writeln** do contain some limited abilities to convert binary data types like **Integer** to their printable representations. So it is in fact legal to **Write** an integer to **Output** because **Write** converts the integer to printable ASCII characters before **Output** ever sees it.

Binary files vs. text files

All disk files can eventually be seen as collections of bytes grouped somehow on a disk or Flash storage device. Pascal makes a distinction between files that can contain absolutely any binary pattern in a stored byte, and those files that are allowed to contain only printable ASCII characters and certain (very few) control characters. Files that may contain any byte pattern at all are called *binary* or *non-text* files. Files that are limited to printable characters are called *text files*.

Text files are allowed to contain printable ASCII characters plus *whitespace* characters. Whitespace characters include carriage return ($0D), linefeed ($0A), formfeed ($0C), horizontal tab ($09), and backspace ($08). Although not technically a whitespace character, the bell character ($07) is also permitted in text files, though nearly all operating systems ignore it. The bell character is a holdover from the ancient days of Teletype terminals, which rang a little bell when the character appeared in their text streams.

An important consequence of allowing only printable characters is that text files created under DOS or Windows may contain an end-of-file (EOF) marker character. Binary files may *not* contain an EOF marker because binary files are allowed to

contain any 8-bit character pattern as valid data. Therefore the operating system (and thus FreePascal) using a binary file could not tell an EOF marker character from just more legal binary data.

There are historical (and now ancient) reasons for this: Text files created under the DOS operating system traditionally used Ctrl-Z (hex 1A) as the EOF marker. This was a holdover from DOS's predecessor CP/M-80, which did *not* keep track of the size of files down to the byte, as DOS did. CP/M-80 only knew how many 128-byte blocks were contained in a file; to mark the actual end of data within the last data block, an application had to place a Ctrl-Z character after the last true byte of data in the file.

Although many older compilers do, FreePascal does not write a Ctrl-Z after the last data byte in a text file. Furthermore, when reading a text file, it reports EOF at *either* the true EOF reported by the operating system *or* at the first Ctrl-Z character found in the file. Although FreePascal does not end a text file with Ctrl-Z, you may at some point have to read a text file into a FreePascal program that was *not* created using FreePascal! This could be important if you allow a Ctrl-Z character to be written somewhere in the middle of a text file, perhaps by way of the **Write** procedure as used for binary data like records, sets, integers, etc. If you later read that file as text through **Read** or **Readln**, you will be able to read only up to the first Ctrl-Z, at which point **EOF** will return **True**.

Text files in FreePascal are declared this way:

```
VAR
  MyFile   : FILE OF Text;      { Both forms are legal }
  YourFile : Text;
```

Any file that is *not* declared to be a **FILE OF Text** or simply **Text** is considered a binary file. The difference between the two becomes critical when you need to detect where data ends in the file. We'll examine this problem in detail in connection with the **EOF** function in Section 14.5

14.4. Reading from and Writing to Files

The original ISO Standard Pascal defined a pair of file procedures called **Get** and **Put**. These worked with a data structure called the *file window,* which is a "slice" of the file to be read or written to. When you **Get** from a file, the next item in the file is placed in the file window, from which it may be accessed at leisure. Similarly, when you wish to write an item to a file, you place it in the file window and then **Put** the file window out to the disk.

FreePascal does *not* implement **Get** and **Put**, nor does it make the file window directly available to the programmer. I mention them here only because you may run into them reading ancient Pascal code.

Using Read and Write with binary (non-Text) files

The following discussion deals with *typed* files, that is, binary files that are not files of **Text**. Text files are a special case and will be covered in the next section.

Once a file has been opened with **Reset**, data may be read from the file with the **Read** procedure. **Read** takes at least two parameters: a file variable and one or more variables with the same type as the file:

```
VAR
   StatRec,Rec1,Rec2,Rec3  : NameRec;
   MyFile : FILE OF NameRec;

Assign(MyFile,'B:NAMES.DAT');
Reset(MyFile);

Read(MyFile,StatRec);
Read(MyFile,Rec1,Rec2,Rec3);
```

Here, **NameRec** is a record type defined earlier in the program. In this example, the **Read** statement reads the *first* record from **MyFile** into the record variable **StatRec**. There is no mention of the file window variable, although the file window is involved beneath the surface in the code that handles the **Read** function. The second **Read** statement reads the next three elements of **MyFile** into **Rec1**, **Rec2**, and **Rec3**, all at one time. This is a convenient shorthand; there is no difference in doing this than in executing three distinct **Read** statements.

Write works just the same way, with the same parameters; the only difference being the direction the data is flowing:

```
Assign(MyFile,'B:NAMES.DAT');
Rewrite(MyFile);

Write(MyFile,StatRec);
Write(MyFile,Rec1,Rec2,Rec3);
```

Here, the file **MyFile** is opened and cleared with **Assign** and **Rewrite**. Then the four defined records are written to the file.

Read and **Write** work sequentially; each time **Read** or **Write** is used, the file

pointer is bumped to the next element down the file. The process never works backwards; you cannot begin at the end of the file and work your way back. For that sort of thing you need random file I/O, which I cover briefly at the end of this chapter.

Using Read and Write with text files

A text file is a stream of printable characters between the program and a device file or disk file. The **Read** and **Write** statements are used to transfer a data item (or list of data items) to or from the text file character stream. The data items do not have to be of the same type. We've been doing things like this informally all through this book:

```
VAR
  Unit  : Char:
  Count : Integer;

Write('The number of files on disk ',Unit,' is ',Count);
```

What is happening here is that a list of four data items is being sent to device file **Output**. Two are string literals, one is a character, and one is an integer.

Text files, as we saw in the previous section, may contain only printable ASCII characters and certain control characters. **Integer** variables are not ASCII characters; rather, they are two-byte binary numbers that are not necessarily printable to the screen.

Yet when you write

```
VAR
  I : Integer;

I := 42;
Write(I);
```

the two ASCII characters "4" and "2" appear on your screen. They are an ASCII representation of a two-byte binary integer that could as well have been written in base 2 as 0000000000101010. **Read** and **Write** contain the machinery for converting between numeric variables (which are stored in binary form) and printable ASCII numerals.

Write also has the ability to take **Boolean** values (which are actually binary numbers 0 or 1) and convert them to the words "True" and "False" before passing

them to a text file. **Read**, however, does not convert the ASCII strings "TRUE" or "FALSE" to Boolean values!

Read and **Write** when used with text files can accept data types **Integer**, **Char**, **Byte**, **Word**, **LongInt**, **SmallInt**, **ShortInt**, **Cardinal**, **Currency**, **QWord** and subranges of those types; type **ShortString** and derived string types; plus type **Real**, **Single**, **Double**, **Extended**, and **Int64**. (**Comp**, while it does work with **Write**, is now considered obsolete and may not reliably work in all FreePascal constructs.) Write will also accept **Boolean** values; remember that **Read** will not. Enumerated types, pointers, sets, arrays, and record types will generate errors during compilation if you attempt to use them with text files.

A quick note about type **Currency**: Although the type was created to handle decimal money values, to show a **Currency** value as dollars and cents (or some other decimal currency), you must use write parameters, with the width after the decimal set to 2:

```
Bux : Currency;
Bux := 337.95
Write(Bux : 5 : 2);
```

When printed using Write or Writeln, you'll see this:

```
337.95
```

As best I know, FreePascal has no feature to automatically place a dollar sign ($) in front of a displayed **Currency** value.

Although **Read** will accept a string variable, it's better practice to use **Readln** for reading strings. (See below on **Readln**.) The problem is that strings on a text file character stream carry no information about how long they are. If you **Read** from a text file stream into a string variable, the string variable will accept characters from the file until it is physically filled. If the file ends before the string is completely filled, the system may hang or fill the remainder of the string with garbage.

Readln and Writeln

Pascal defines a pair of I/O statements that work only on text files: **Readln** and **Writeln**. They introduce a whole new concept: dividing a Pascal text file into "lines."

Ordinary text on a printed page is a series of lines. A Pascal text file is a one-dimensional stream of characters; it has none of the two-dimensional quality of a printed page. To model text as we see it in the real world, Pascal defines an end-of-

line character (EOL) that is inserted into the character stream of a text file after each line of printable characters. The definition of the EOL character may vary among operating systems. For most microcomputers EOL is not one character but two: the sequence carriage return/line feed ($0D/$0A). This is a holdover from Teletype's heyday when it took one control character to return the typehead to the left margin, and yet another to index the paper up one line.

The **Writeln** procedure is exactly like **Write**, save that it follows the last item sent to the text file character stream with the EOL character. For our discussion EOL will always be the pair CR/LF.

```
PROGRAM WriteInt1;

VAR
  IntText : Text;
  I       : Integer;

BEGIN
  { Create 'D:\Temp\' or change it to a valid drive or }
  { path on your system: }
  Assign(IntText,'D:\Temp\INTEGERS.TXT');
  Rewrite(IntText);
  FOR I := 1 TO 25 DO Writeln(IntText,I);
  Close(IntText);
END.
```

This program **WriteInt1** writes the ASCII equivalent of the numbers from 1-25 to a text file named INTEGERS.TXT. Each numeral is followed by a CR/LF pair. A hexdump of file INTEGERS.TXT will allow you to inspect the file character stream:

```
0000  31 0D 0A 32 0D 0A 33 0D    0A 34 0D 0A 35 0D 0A 36
0010  0D 0A 37 0D 0A 38 0D 0A    39 0D 0A 31 30 0D 0A 31
0020  31 0D 0A 31 32 0D 0A 31    33 0D 0A 31 34 0D 0A 31
0030  35 0D 0A 31 36 0D 0A 31    37 0D 0A 31 38 0D 0A 31
0040  39 0D 0A 32 30 0D 0A 32    31 0D 0A 32 32 0D 0A 32
0050  33 0D 0A 32 34 0D 0A 32    35 0D 0A
```

If you know your ASCII well (or have a table handy on paper or online) you can see the structure of the character stream in this file: ASCII numerals separated by $0D/$0A pairs. Because Windows knows to the byte (here, 91) how large the file is, it doesn't need to tack a Ctrl-Z on the end of the file. Older Pascals running on older operating systems may fill out the remainder of the file's 128-byte block with Ctrl-Zs.

In a sense, **Writeln** writes only strings (minus length bytes) to its files. If you hand it a variable that is not a string, **Writeln** will (if the conversion is possible) convert it to its string equivalent before writing it to its file.

Readln reads one line from a text file. A line in a text file, again, is a series of characters up to the next EOL marker. If **Readln** is reading into a string variable, the number of characters read becomes the logical length of the string. So unlike **Read** and **Write**, **Readln** and **Writeln** can in fact maintain information on string length in a file, since all strings written by **Writeln** are terminated by EOL markers.

Formatting with write parameters

Write and Writeln allow certain formatting options when writing data to text files, including the screen (**Output**) and system printer. These options as given to the Write and Writeln statements are called *write parameters*.

The simplest write parameter applies to any variable that may be written to a text file, and specifies the width of the field within which the variable is written.

```
<any writeable variable> : <field width>
```

When written, data (if any) in the variable will be right-justified within a field of spaces <field width> wide. For example:

```
CONST
  Bar = '|';

VAR
  I    : Integer;
  R    : Real;
  CH   : Char;
  OK   : Boolean;
  Txt  : String;

I := 727;
CH := 'Z';
R := 2577543.67;
OK := False;
Txt := 'Grimble';


Writeln(Bar,I:5,Bar);
Writeln(Bar,CH:2,Bar);
Writeln(Bar,R:12,Bar);
Writeln(Bar,OK:7,Bar);
Writeln(Bar,TXT:10,Bar);
Writeln(Bar,-R,Bar);
Writeln(Bar,R,Bar);
```

When run, the above code snippet produces this output:

```
|   727|
| z|
| 2.5775E+006|
|   FALSE|
|    Grimble|
|-2.5775436699999999E+006|
| 2.5775436699999999E+006|
```

Note that the real numbers are always expressed in exponential (also called "scientific") notation, that is, as powers of ten, even though originally expressed with a decimal point and no exponent. To express a real number without the exponent, you must include a second write parameter for the width of the decimal part of the field:

```
<real value> : <field width> : <decimal width>
```

The value <decimal width> indicates how many decimal places are to be displayed. For example:

```
R := 7.775;
S := 0.123456789;
T := 7765;

Writeln(Bar,R:10:3,Bar);
Writeln(Bar,R:10:1,Bar);
Writeln(Bar,R:5:2,Bar);
Writeln(Bar,S:6:6,Bar);
Writeln(Bar,S:12:6,Bar);
Writeln(Bar,S:12:12,Bar);
Writeln(Bar,S:5,Bar);
Writeln(Bar,T:5:2,Bar);
Writeln(Bar,T:5,Bar);

Writeln(Bar,T:6:6,Bar);
```

This code produces the following output:

```
|     7.775|
|       7.8|
| 7.78|
|0.123457|
|    0.123457|
|0.123456789000|
| 1.2E-001|
```

```
|7765.00|
| 7.8E+003|
|7765.000000|
```

A reminder here: You cannot begin a fractional real number (such as .123456789) with a decimal point. You *must* begin the number with a 0 as shown, or FreePascal will issue an error during compilation.

Writing to the system printer

Writing to the system printer under Lazarus is complicated, and involves packages, a feature I don't cover in this book. The process is very different depending on whether you're printing from a Windows console or from a GUI program using components in the Lazarus Component Library (LCL). There is a solid tutorial on the FreePascal wiki, and I recommend reading it closely and investigating how packages are used from within programs:

<p align="center">https://wiki.freepascal.org/Using_the_printer</p>

14.5. IOResult and EOF

IOResult

Working with creatures like disk files that lie outside the borders of the program itself is risky business. You can build machinery into your program to make sure that the program never attempts to divide by zero, or never attempts to index past the bounds of an array. But how do you make sure that (when you want to open a disk file) the file is on the disk and the disk is in the proper disk drive?

The program does not have absolute control over disk files in the way it has absolute control over numbers, arrays, and other variables. The only way to be sure a file is on the disk is to go out and try to read it. If the file isn't out there, you need some way to recover gracefully.

The runtime code that FreePascal adds to every program it compiles guards against runtime errors such as an attempt to open (for reading) a file that does not exist. If such an error is detected,your program will terminate.

Obviously, if there is a legitimate possibility that a file does not exist on the disk, you cannot allow your program to just crash. Better to determine that an error has happened without crashing, so that the program could do something about it, like create a new file or look somewhere else for the old one. FreePascal provides the

IOResult function to let the program know how successful it has been in striking a path to the outside world. It is predeclared by FreePascal this way:

```
FUNCTION IOResult : Integer
```

After each I/O statement is executed, a value is given to the **IOResult** function. This value can then be tested to determine whether the I/O statement completed successfully. A 0 value indicates that the I/O operation went normally; anything else constitutes an error code.

However, the runtime code's error traps will still crash your program when an error is encountered, whether or not you use **IOResult**. To keep the program running in spite of the error, you must disable the error trap with the compiler directive **$I**. This is done by surrounding the I/O statement with an **{$I-}** directive (turn traps off) and an {$I+} directive (turn traps on again.)

For example:

```
Assign(MyFile,'D:BOWLING.DAT');
{$I-} Reset(MyFile); {$I+}  { Suspend error traps during Reset }
IF IOResult <> 0 THEN
  BEGIN
    Beep;
    Writeln('>>The bowling scores file cannot be opened.');
    Writeln('  Make sure the scores file is on the D: drive');
    Writeln('  and press (CR) again:')
  END;
```

This code snippet checks **IOResult** immediately after executing a **Reset** to open a file for read. If **IOResult** returns a non-zero value, the program displays a message to the operator.

IOResult is cleared to zero and refilled with a new value at the beginning of *all* I/O primitives. This includes **Readln** and **Writeln** when used (as we have been using them all along) to talk to the keyboard and the screen via device files **Input** and **Output**. This means you *cannot* directly write the value returned by **IOResult** to the screen or to a file! So don't do this:

```
{$I-} Reset(MyFile); {$I+}
Writeln('The result of that Reset is ',IOResult);  { No! }
```

No matter what happens when **Reset(MyFile)** is executed, the above snippet will always display:

```
The result of that Reset is 0
```

Given that restriction, if you want to see the actual value returned by **IOResult**, you must assign the value of IOResult to an integer variable immediately after an I/O statement and work with the integer variable rather than the IOResult function itself. Remember: **IOResult** is a *function,* not a variable!

Another example of **IOResult** in use lies in the Averager program in the next section. Averager opens a binary file of integers, reading and displaying them one-by-one until it reaches the end of the file, and then calculates and displays the average of the integers it has read. If it cannot open the file of integers it recovers and issues an error message with the help of **IOResult**.

EOF

Knowing where a disk file ends is critical. A built-in function called **EOF** (End Of File) provides this service to your programs. **EOF** is predeclared this way:

```
FUNCTION EOF(FileVar : <any file type>) : Boolean
```

FileVar can be any legal file type. The **EOF** function returns **True** as soon as the last item in the file has been read. At that point the file pointer points just past the end of data in the file, and no further reads should be attempted. A runtime error will occur if you try to read beyond the end of a file. This applies to both text files and binary files.

FreePascal's runtime code determines **EOF** by using information stored by the operating system for each disk file. For text files it keeps a count of characters read from the file and compares that to the size of the file at each read; when bytes read equals or exceeds the number of bytes the OS says are in the file, **EOF** returns **True**. For binary files, FreePascal knows the size of each record read (since every file is a **FILE OF <something>**, where **<something>** is a declared data type with a fixed size) and compares a similar count of records read against the file size figure.

Text file EOF is also triggered by the presence of a Ctrl-Z character ($1A) in the file. Any modern OS knows exactly how many bytes are in every file at all times, so with text files, FreePascal reports EOF either when the OS indicates the last character had been read, or when a Ctrl-Z is encountered in the text stream from the file.

The following program assumes a binary file of 16-bit small integers (type **SmallInt**) named INTEGERS.BIN. (I provide such a file on the listings archive for this book, or if you're ambitious you can write a short program that generates a file for you. Try it!) Averager reads and displays all the integers in the file, testing **EOF** at each read. As it reads each integer it keeps a running total and a running count, and then produces an average value for all the integers in the file. Note the use of the **$I-** and **$I+** compiler directives when opening the file:

```
{-------------------- ---------------------------------------}
{                          Averager                          }
{            Binary file I/O demonstration program           }
{                          by Jeff Duntemann                 }
{                          FreePascal 3.2.2                  }
{                          Last update 5/3/2025              }
{                                                            }
{      From: FREEPASCAL FROM SQUARE ONE  by Jeff Duntemann   }
{------------------------------------------------------------}


PROGRAM Averager;


VAR
  IntFile      : FILE OF SmallInt;
  I,J,Count    : SmallInt;
  Average,Total : Real;


BEGIN
  Assign(IntFile,'INTEGERS.BIN');  { Open the file of 16-bit integers }
  {$I-} Reset(IntFile); {$I+}      { Traps off, Reset, then on again  }
  I := IOResult;
  IF I <> 0 THEN                   { If IOResult isn't 0, can't open it }
    BEGIN
      Writeln('>>File INTEGERS.BIN is missing or damaged.');
      Writeln('  Please investigate and run the program again.');
      Write('  Press Enter:');
      Readln;
    END
  ELSE
    BEGIN
      Count := 0; Total := 0.0;     {Initialize variables}
      WHILE NOT EOF(IntFile) DO      {Go through the file }
        BEGIN
          Read(IntFile,J);              {Read a number from the file    }
          IF NOT EOF(IntFile) THEN {If EOF is not set, keep going  }
            BEGIN                        {Display integer & add to total }
              Writeln(J:6);
              Count := Count + 1;
              Total := Total + J
            END;
        END;
      Close(IntFile);               { When it hits EOF, close the file }
      Average := Total / Count;     { Calculate average                }
      Writeln;                      { Show count & average values:     }
      Writeln('>>There are ',Count,' integers in INTEGERS.BIN.');
      Writeln('  Their average value is ',Average:10:4,'.');
      Write('  Press Enter:');      { Wait for user to press Enter     }
      Readln;
    END
END.
```

14.6. FileSize & FilePos

FreePascal supplies two built-in functions for determining the size of a binary disk file and the position of the file pointer within a disk file. Note that **FileSize** and **FilePos** may *not* be used with text files!

FileSize is predeclared this way:

```
FUNCTION FileSize(FileVar : <binary filetype>) : Int64;
```

The function **FileSize** returns an **Int64** count of the number of items stored in the file. **FileVar** is an open binary file. (*Not* a text file!) An empty file will return a zero value.

Function **FilePos** is predeclared this way:

```
FUNCTION FilePos(FileVar : <binary filetype>) : Int64;
```

A binary file's file pointer is a counter that indicates the next item to be read from the file. The **FilePos** function returns the current value of a file's file pointer. **FileVar** is an open binary file. (*Not* a text file.) The first item in a file is item number 0. When a file is first opened, its file pointer is set to 0. Each **Read** operation will increment the file pointer by one. The **Seek** procedure (more on **Seek** later in this chapter) will put the file pointer to a particular value to enable random access to a binary file.

14.7. Miscellanous File Routines

The last group of FreePascal file routines we'll discuss are all extensions to the ISO Standard Pascal definition. Most Pascals have routines like these, but sadly, their invocation syntax and parameters vary widely. Any time you use file I/O, you can be almost certain that your programs will be non-portable. (There is no way even to close a file in ISO Pascal!) This is yet another reason why I have not stressed adherence to the ISO Standard in this book. If file I/O cannot be made portable, you might as well hang it up.

Flush

Every file has a buffer in memory, and when you write to a file, the data you've written actually goes to the buffer rather than directly to the physical disk file. Periodically, based on decisions it makes on its own, the Turbo Pascal runtime code will flush the buffer to disk, actually transferring the data to the physical disk file. You can force such a flush to disk with the Flush procedure. It is predeclared this way:

```
PROCEDURE Flush(<filevar>);
```

Here, **<filevar>** is any file variable opened for output. **Flush** has no effect on a file opened for input. **IOResult** will return a 0 value if the file was flushed successfully. *Do not use Flush on a closed file!*

Close

Closing a file that you have opened ensures that all data written to the file is physically transferred from the file buffer to disk. In FreePascal the **Close** procedure does this job:

```
PROCEDURE Close(<filevar>);
```

As with **Flush**, **<filevar>** is any opened file. It is legal to use **Close** to close a file that has been closed already, though it doesn't accomplish anything useful.

And of course, if you exit a program before closing a file that has been written to, the runtime code makes no guarantee that all records written to the file buffer will actually make it out to the physical disk file. As with most file-related routines, **IOResult** will be set to 0 if the file was closed successfully. Otherwise, **IOResult** will contain a nonzero error code.

Erase

Deleting a disk file from within a program is done with the Erase procedure:

```
PROCEDURE Erase(<filevar>);
```

Again, **<filevar>** is any file variable that has been assigned to a physical file. In other words, if you try to **Erase** a file variable to which no physical file has yet been assigned, the runtime code has no way of knowing which file you want to delete:

```
VAR
  NumFile : FILE OF Integer;

Assign(NumFile,'VALUES.BIN');
Erase(NumFile);
```

Do *not* attempt to **Erase** an open file. Close it first, or the results could be unpredictable. **IOResult** will return 0 if the file was successfully deleted.

Rename

FreePascal allows you to change the name of a disk file from within a program with the Rename procedure:

```
PROCEDURE Rename(<filevar>; NewName : ShortString);
```

\<filevar\> is any file that has been assigned to a physical file with **Assign**. As with **Erase**, trying to rename a file without connecting it with a physical file is meaningless. Also like **Erase**, renaming an open file is a no-no. **IOResult** will return a 0 if the file was successfully renamed.

Append

Append provides a way to quickly move the file pointer to the end of a text file without having to read the file and throw away the characters read up to EOF.

Append is used instead of **Rewrite**:

```
PROCEDURE Append(<filevar>);
```

\<filevar\> must first be assigned to some physical file before **Append** can be used. The contents of the file are not destroyed, as they are with **Rewrite**. The file is opened for output, however, at EOF. Adding text to the file with **Write** or **Writeln** will position the new text at and following EOF.

IOResult will return a 0 if the operation was successful. Keep in mind that **Append** may be used *only* with text files. Attempting to use **Append** with a non-text file type will trigger

```
Error 63: Invalid file type
```

Truncate

Truncate is conceptually similar to **Append**. Both prepare a file for the adding of additional data via **Write** or (for text files) **Writeln**. Unlike **Append**, **Truncate** may be used with any type of file: text, binary, or untyped. **Truncate** is predeclared this way:

```
PROCEDURE Truncate(<filevar>);
```

When executed, **Truncate** chops a file off at the current position of the file pointer. In other words, if you have read part way down a file and execute **Truncate**, the remainder of the file will be thrown away. The file is then ready for output, even if you opened the file with **Reset**.

14.8. Using text files

Given its variable-length string type, its built-in string functions and procedures, and the **Readln** and **Writeln** file procedures, FreePascal is a natural choice for working with text files. In this section we'll show you a real-life example of a useful program for manipulating text files.

Filter programs

There is a whole class of programs that read a file in chunks, perform some manipulation on the chunks, and then write the transformed chunks back out to another file. This type of program is called a "filter" program because it filters a file through some sort of processing step. The data changes according to a set of rules as it passes through the processing part of the program.

A good example would be a program to force all lower-case characters in a file to upper case. FreePascal is not sensitive to character case but some programs and language processors (generally old versions of old languages like COBOL) do not interpret lowercase characters correctly. To pass a text file between Pascal and vintage COBOL, all lowercase characters in the file must be set to uppercase.

A filter program to accomplish this task would work this way:

Open the input file for read and create a new output file.
While not end-of-file keep doing this:
    Read a line from the input text file.
    Force all lowercase characters in the line to uppercase.
    Write the line out to the output text file.
  Close both files.

This basic structure is the same for all text file filter programs, except for the processing that is done. You could just as easily force all uppercase characters in the line to lowercase, count the words in the line, remove all BEL characters (Ctrl-G) from the line, expand HT (tab; Ctrl-I) characters to 8 space characters, and so on.

You could in fact combine two or more processes into one filter program; say, force lowercase to uppercase and count characters.

The program Caser shown below can perform two distinct functions: It can force all lowercase characters in a text file to uppercase, or all uppercase characters to lowercase. (Not both at the same time, however.) Which of the two actions is taken depends on a parameter entered on the command line:

```
Caser up D:COBOL.SRC        Forces lower to upper

Caser down D:PASTEXT.SRC    Forces upper to lower
```

Caser incorporates a good many of the file routines we've been discussing in this chapter, and understanding how it works will help you understand how those routines interact within a program.

```
{---------------------------------------------------------------}
{                           Caser                               }
{                                                               }
{ An upper/lower case conversion filter program for text files  }
{                                                               }
{                                  by Jeff Duntemann            }
{                                  FreePascal V3.2.2            }
{                                  Last update 5/4/2025         }
{                                                               }
{      From: FreePascal From Square One  by Jeff Duntemann      }
{---------------------------------------------------------------}


PROGRAM Caser;

CONST
  Upper = True;
  Lower = False;

VAR
  Quit      : Boolean;
  WorkFile  : Text;
  TempFile  : Text;
  NewCase   : Boolean;
  WorkLine  : ShortString;
  WorkName  : ShortString;
  TempName  : ShortString;
  CaseTag   : ShortString;


  {>>>>ForceCase<<<<}

  FUNCTION ForceCase(Up : BOOLEAN; Target : STRING) : STRING;

  CONST
    Uppercase : SET OF Char = ['A'..'Z'];
    Lowercase : SET OF Char = ['a'..'z'];

  VAR
    I : INTEGER;

  BEGIN
    IF Up THEN FOR I := 1 TO Length(Target) DO
      IF Target[I] IN Lowercase THEN
        Target[I] := UpCase(Target[I])
      ELSE { NULL }
    ELSE FOR I := 1 TO Length(Target) DO
      IF Target[I] IN Uppercase THEN
        Target[I] := Chr(Ord(Target[I])+32);
    ForceCase := Target
```

```
   END;



{>>>>MakeTemp<<<<}

PROCEDURE MakeTemp(FileName : ShortString; VAR TempName : ShortString);

VAR
  Point : Integer;

BEGIN
  Point := Pos('.',FileName);
  IF Point > 0 THEN Delete(FileName,Point,(Length(FileName)-Point)+1);
  TempName := Concat(FileName,'.$$$')
END;



{ Caser main program: }

BEGIN
  Quit := False;
  IF ParamCount < 2 THEN    { Missing parms error }
    BEGIN
      Writeln('>>CASE<<  V2.00  By Jeff Duntemann');
      Writeln('           From the book, FREEPASCAL FROM SQUARE ONE');
      Writeln('           Last modified 5/1/2025');
      Writeln;
       Writeln('This program forces all characters of a text file to either ');
      Writeln('upper or lower case, as requested.  Characters already in ');
      Writeln('the requested case are not disturbed.');
      Writeln;
      Writeln('CALLING SYNTAX:');
      Writeln;
      Writeln('CASE UP|DOWN <filespec>');
      Writeln;
      Writeln('For example, to force all lowercase characters of file');
      Writeln('FOO.COB to uppercase, invoke CASE this way:');
      Writeln;
      Writeln('CASE UP FOO.COB');
      Writeln;
    END
  ELSE
    BEGIN
      WorkName := ParamStr(2);
      Assign(WorkFile,WorkName);  { Attempt to open the file }
      {$I-} Reset(WorkFile); {$I+}
      IF IOResult <>0 THEN
        BEGIN
          Writeln('<<Error!>> File ',WorkName,' does not exist.');
```

```
            Writeln('           Invoke CASE again with an existing FileName.');
            END
       ELSE
         BEGIN                      { See if UP/DOWN parm was entered }
           CaseTag := ParamStr(1);
           CaseTag := ForceCase(Upper,CaseTag);
           IF CaseTag = 'UP' THEN NewCase := Upper ELSE
             IF CaseTag = 'DOWN' THEN NewCase := Lower ELSE
               Quit := True;
           IF Quit THEN
             BEGIN
               Writeln
               ('<<Error!>> The case parameter must be "UP" or "DOWN."');
               Writeln
               ('           Invoke CASE again using either "UP" or "DOWN".');
             END
           ELSE
             BEGIN
               Write('Forcing case ');
               IF NewCase THEN Write('up ') ELSE Write('down ');
               MakeTemp(WorkName,TempName);  { Generate temporary FileName }
               Assign(TempFile,TempName);    { Open temporary file }
               Rewrite(TempFile);
               WHILE NOT EOF(WorkFile) DO
                 BEGIN
                   Readln(WorkFile,WorkLine);
                   Write('.');                { Dot shows progress }
                   WorkLine := ForceCase(NewCase,WorkLine);
                   Writeln(TempFile,WorkLine)
                 END;
               Close(TempFile);              { Close the temporary file }
               Close(WorkFile);              { Close original source file... }
               Erase(WorkFile);              { ...and delete it. }
               Rename(TempFile,WorkName);    { Temporary file becomes source }
             END
         END
     END
   END
END.
```

Most of Caser is setup: making sure files exist; making sure valid commands were entered, and so on. The real meat of the program is simplicity itself:

```
WHILE NOT EOF(WorkFile) DO
  BEGIN
    Readln(WorkFile,WorkLine);
    Write('.');                      { Dot shows progress }
    SetCase(ForceCase,WorkLine);
    Writeln(TempFile,WorkLine);
  END;
```

This loop executes repeatedly as long as there are lines to be read in WorkFile. A line is read, ForceCase adjusts the case of the characters in the line, and then the line is written to TempFile. When Readln reads the last line in the text file, the EOF function will immediately return True. The text file is "filtered" through ForceCase into a temporary file. When the original file has been read completely, it is erased with Erase and the temporary file is renamed to become the original file.

If you're nervous about deleting your original text file (and that is not a totally unhealthy feeling) you could close it with Close instead of Erase and then give TempFile a new file extension instead of .$$$. (I have used ".ZZZ" in the past.) If you're careless about backing up important files, this is a very good idea.

14.8. Using Binary Files

A text file is distinctive in being the only type of Pascal file that may contain records of varying lengths, if each line is considered a record. Pascal treats text files specially in other ways, by limiting the range of characters that may legally reside in the file, and by the EOF character (on some platforms) that flags where written data ends. Binary files, by contrast, are given none of this special treatment.

A binary file is a file containing some number of data items of a given type:

```
TYPE
  KeyFile = FILE OF KeyRec;
  CfgFile = FILE OF CfgRec;
  IntFile = FILE OF Integer;
```

Only one data type may be stored in a binary file. You could not, for example, write one variable of type **KeyRec** and another variable of type **CfgRec** to the same logical file.

Each instance of a data item in a binary file is called a "record," whether the data item is actually a Pascal **RECORD** type or not. One integer stored in an **IntFile** as defined above could be considered a record of the **IntFile**.

One common use of binary files puts only one record in the file: The "configuration file." Consider a complicated program that performs file maintenance and telecommunications. The program is used at a great many sites owned by a large corporation. The names of the files it works with change from site to site, and so on

Rather than "hard-code" things like telephone numbers into the Pascal source file, it makes more sense to store them out to a configuration file. The easiest way is to define a record type containing fields for all the site-specific information:

```
TYPE
  CfgRec  = RECORD
               SiteName   : String;
               SiteCode   : Integer;
               AuthOp     : String;
               HostCode   : Integer;
               PhoneNum   : String;
               P1FileName : String;
               P2FileName : String;
               AXFileName : String
            END;

  CfgFile = FILE OF CfgRec;

VAR
  SiteFile : CfgFile;
  CfgData  : CfgRec;
```

All the data items that change from site to site are present in one single record. When loaded with the correct site values for a particular site, the record can be easily written to disk:

```
Assign(SiteFile,'B:SITEDATA.CFG');
Rewrite(SiteFile);
Write(SiteFile,CfgData);
Close(SiteFile);
```

Here, once **SiteFile** is opened, the single **CfgRec** with changed data is written to disk with the **Write** statement.

Random file access

So far we've looked at files being read from and written to sequentially. FreePascal also has procedures and functions for dealing with records in files "at random," meaning your code can move the file pointer to any record in the file, and read or write the record at the location specified by the file pointer.

As I explained earlier in this chapter, FreePascal supplies two built-in functions for determining the size of a binary disk file and the position of the file pointer within the file. Remember that **FileSize** and **FilePos** may *not* be used with text files!

A binary file's file pointer is a counter that indicates the next item to be read from the file. The **FilePos** function returns the current value of a file's file pointer. The first item in a file is item number 0. When a binary file is first opened, its file pointer is set to 0. Each **Read** operation thereafter will increment the file pointer by 1.

If you know the position of a record in a file, you can move the file pointer to that record in one operation by using **Seek**. The **Seek** procedure allows you true random access to the file. Seek is predefined this way:

```
PROCEDURE Seek(FileVar : <binary filetype>, RecNum : Int64)
```

Attempting to seek beyond the end of a binary file will trigger a runtime error. Testing a file's size using the **FileSize** function before using **Seek** is always a good idea.

Databases are better!

Given the binary file procedures and functions that come with FreePascal, you might be tempted to build what amounts to a database. Although that might be an effective learning project, if you really need database functionality, keep in mind that FreePascal and Lazarus contain some very powerful components for building databases. Those components do all the tricky work for you. Also, you'll learn about relational database structures and the Structured Query Language (SQL), both of which lie at the heart of modern database development.

Again, as for all GUI programming using Lazarus, database components are objects, and I don't have room in this book to go into either objects or the principles of object-oriented programming. Another book, this one focusing on objects, the Lazarus GUI builder, and its components, is in the works.

CHAPTER 15.
UNITS AND SEPARATE
COMPILATION

Why compile the whole thing when you only need to compile the piece you're working on? Time is money (even small slivers of time add up) and compilation takes time. Why waste that time?

There's no need to. Pascal naturally separates a program into logical chunks—at least if you don't fight the spirit of the language. Good program design calls for relatively independent modules that may be compiled separately, and then linked together in one quick, final step before testing the completed program.

This is called *separate compilation*. Separate compilation has never been part of the official Pascal language definition, but time has shown that it's very hard to manage large projects without it. To provide for separate compilation, FreePascal and most others (including the Borland Pascals) implement the *units* paradigm pioneered by UCSD Pascal decades ago. This chapter introduces the mechanisms by which separate compilation happens in FreePascal. It's a surprisingly subtle business, and not all of it can be covered in an introductory text such as this. I'll have more to say about separate compilation and units in future books on FreePascal and Lazarus.

15.1. THE PACKING LIST METAPHOR

Let's say the UPS man rolls up to your door one day and drops a cardboard box from Lieutenant Kije's Military Surplus in your porch. You know where it came from, but your memory of what the order contains has gotten a little fuzzy.

So you rip open the little clear plastic slap-on pocket and pull out the sheet of paper marked "packing list." Right in a row is a summary of what's in the box: 2 spur gears, 96 tooth, brass. One pillow block ball bearing, 1/4". One alarm clock, Navy Surplus. (Original cost $900.) One tank prism; tank not included.

Without actually ripping open the box, you then know what's in it. If, for example, the packing list had read something like, "25 polyethelene shower curtains, mauve," I would start to suspect that the UPS man had dropped the wrong box on my porch.

A Pascal unit is like a little like a sealed box with a packing list. Each unit has two primary parts:

1) an interface part; and

2) an implementation part.

The interface part is a lot like a packing list. It is an orderly description of what is in the unit, without the actual code details of the functions and procedures within the unit. The interface part may include constant, type, and variable definitions, along with the parameter line portions of the functions and procedures within the unit.

This last item may seem a little strange. Consider the following line of code:

```
PROCEDURE FogCheck(InString : String; VAR FogFactor : Integer);
```

This isn't all of the procedure, obviously—but like it or not, it's all you really need to see of **FogCheck** to be able to make use of the procedure in your own programs. You still need to know the relationship between the input parameter **InString** and the output parameter **FogFactor**, of course, but that's a documentation issue. Knowing what procedure **FogCheck** does is an entirely separate matter from knowing how it does it. If you know that the foggier the input string is, the higher the value will come back in **FogFactor**, well, that's sufficient in most cases.

Where's the rest of **FogCheck**? In the implementation part of the unit. In other words, inside the sealed box. The box contains the substance of the order, the actual goods. The packing list contains a description. That is the critical difference between interface and implementation.

At this point the packing list metaphor begins to break down, because in order to do anything with my brass gears and tank prism I have to rip open the box and take the goods out. A separately-compiled unit may remain a sealed box in the sense that you cannot read the details of what lies inside, but the interface part of the unit allows your own programs to hook into and use the contents of the box.

15.2. Using Units

FreePascal comes with numerous precompiled units full of routines for use in your own programs. This works to your advantage in many ways, not the least of which is that the procedures and functions within those readymade units are already compiled, and do not need to be compiled again every time you compile your own programs. Without having to recompile the units that you use, compilation wastes a little less time—and the longer your programs are, the less time compilation wastes.

How do you use these readymade units? Just like that—you USE them:

```
PROGRAM Caveat;

USES Crt;

BEGIN
  ClrScr;                      { In unit Crt }
  GoToXY(12,10);               { ditto }
  Writeln('Better to light one single candle...');
  Writeln('...than to trip on a rake while changing the fuse.');
END.
```

Here, the program **Caveat** contains a new type of Pascal statement: The **USES** statement. (**USES** is a reserved word.)

Caveat uses a unit called **Crt** that is included with FreePascal. **Crt** contains (as you might imagine) routines that deal with screen handling for DOS-style text windows. **ClrScr** and **GotoXY** are the most common examples. Such routines were never really part of the Pascal language. Both are ordinary procedures that you could have written yourself. They originated in UCSD Pascal and have appeared in nearly all other Pascal implementations since then.

I have to point out something important here: The same isn't true about those other stalwarts of simple console-mode example programs, **Read**, **Readln**, **Write**, and **Writeln**. These are not procedures in the strictest Pascal sense. If you've been through the earlier parts of this book you'll know why: They can take a variable number of parameters of many different types in any order at all, which is a gross violation of Pascal's rules and regulations regarding procedures. This being the case, they must be "special cases" built into the compiler, because the compiler has to generate different object code to perform each separate call to Read or Write depending on the number and types of the parameters used. So while many beginners think of Writeln as a CRT-oriented procedure, it does not "live" in the same unit with all the other CRT-oriented functions and procedures supplied with FreePascal.

The **USES** statement can take (almost) any number of unit names, separated by commas. (FreePascal programs are limited to 1,024 units.) The order you place them in the **USES** statement is not important unless code inside one unit references declarations made inside one of the other units. In that case, in keeping with Pascal's dictum of "define it before you reference it" the called identifier must be named *before* the caller. Here's a very simple example of how you can go wrong:

```
PROGRAM DiceGame;    { This won't compile! }

USES Crt,DiceUnit,BoxUnit;

BEGIN
END.
```

The short code snippet above is the top-level structure of a simple dice game. **DiceUnit** draws dice. **BoxUnit** draws boxes. Drawing dice requires drawing boxes. The game draws a box, and then puts some number of "o" characters on the box to draw the faces of typical 6-sided dice.

Clearly, the game has to draw a box before putting the spots on the box. **DiceUnit** calls procedures inside **BoxUnit**. This means that FreePascal has to encounter **BoxUnit** before it encounters **DiceUnit**—and the **Crt** unit before it encounters either. (The procedure **GotoXY** lives in the **Crt** unit, and both of the other units call **GotoXY**.) Rearranging the order in which the units appear in the **USES** statement fixes the problem:

```
PROGRAM DiceGame;    { This will compile! }

USES Crt,BoxUnit,DiceUnit;

BEGIN
END.
```

Don't USE the System unit!

In every program compiled by FreePascal, there is an implicit use of a crucial library called **System**. In other words, the code inside **System** is *always* linked into your executable programs. Do not put **System** in a **USES** clause! FreePascal will post an error message if you do.

Where units should be placed on your disk

Compiled units must exist where FreePascal can find them. This can be a complicated business for reasons I can't explain in a book for newcomers. When the compiler encounters a unit in a **USES** clause, FreePascal will look in these places, in this order:

1. It will look in the current directory; that is, the directory shown when you bring up a console window.

2. It will look in the directory where the unit's source code files exists; in other words, in the project's directory. As a beginner, this is generally the best thing to do, and I'll return to this point shortly.

3. It will look in the directory where the FreePascal compiler's executable file is. This is not a good strategy for beginners!

4. It will look in all directories in the unit search path.

Unfortunately, you can't place a drive specifier or a path specifier in a **USES** clause. In other words, these are not legal **USES** clauses:

```
USES Crt,DOS,C:RingBuf;        { Won't compile! }

USES Crt,D:\JIVETALK\RINGBUF;   { Won't compile! }
```

Unit locations are set on a per-project basis. Each project can have a file path of its own to direct the compiler to the units that the project **USES**. This can be set from the **Project|Project Options|Compiler Options|Paths**. However, remember that when you create and save a new project, Lazarus will keep the directory in which the new project was saved, and look there first for unit files.

Referencing identical identifiers in different units

It's perfectly legal to have identical identifiers within two units and use both units from the same program. In other words, you could have a unit called **CustomCRT** that contained a procedure called **ClrScr**, which is the same name as the familiar screen-clearing routine found in standard unit **Crt**:

```
USES Crt,CustomCrt;
```

A program could use both units as shown above and no error message would be generated. But—which **ClrScr** would be actually incorporated into the program?

With no more information than the procedure name to go on, FreePascal will link the *last* procedure named **ClrScr** that it finds in scanning the units named in the **USES** statement. In the example above, it would scan **CustomCrt** after scanning **Crt**, and thus it would link the custom-written **ClrScr** routine into the program.

You could exchange the unit names **Crt** and **CustomCrt** in the **USES** statement, and the compiler would then link the standard **ClrScr** routine into your program:

```
USES DOS,CustomCrt,Crt;
```

However, if you arrange the **USES** statement like this, nothing in **CustomCrt** can use any of the many useful routines in **Crt**. This may not always be an issue, but it does limit your options.

There is a better way. You can specify the name of the unit that contains an identifier when you use the identifier. The notation should be familiar to you from working with Pascal record types, and works in a very similar fashion:

```
PROGRAM WeirdTextStuff;

USES Crt,CustomCrt;

BEGIN
  Crt.ClrScr;           { Clears the visible PC text screen }
  ClrScr;               { Clears the other text screens too }
  .  .  .
END.
```

This is often called *dotting*. In the same way that you can have two different record types with identical field names, you can have two or more units containing identical identifiers, and there will be no conflict. You simply choose the one you want by prefixing it with the unit name and a period character at each invocation. The default identifier in cases where no unit name precedes the reference is the first one found in scanning the units in the **USES** statement.

Note that the resemblance to record references ends there. FreePascal has no **WITH** statement feature for specifying unit names.

15.3. Unit Syntax

In its simplest form, a unit source code file is very much like a Pascal program source code file without a program body. Typically, units contain procedures and functions, and often other declarations like constants, types, and variables.

A minimal unit with nothing inside it looks like this:

```
UNIT Skeleton;

INTERFACE

IMPLEMENTATION

END.
```

There are some immediate departures from the expected here. The reserved words **INTERFACE** and **IMPLEMENTATION** are not statements, and therefore are not followed by semicolons. Like the reserved words **BEGIN** and **END**, they serve to set off groups of statements that belong together. Also, there is an **END** but no **BEGIN**. Units do not have program bodies.

Fleshing out our unit a little bit will bring out the differences between the interface and implementation parts:

```
UNIT Skeleton;

INTERFACE

USES DOS, Crt;


TYPE
  MyType = ItsDefinition;

VAR
  MyVar : MyType;


PROCEDURE MyProc(MyParm : MyType);

FUNCTION MyFunc(I,Y : Integer) : Char;


IMPLEMENTATION

VAR
  PrivateVar : MyType;


PROCEDURE MyProc(MyParm : MyType);

VAR Q,X : Integer;

BEGIN
END;

FUNCTION MyFunc(I,Y : Integer) : Char;

BEGIN
END;

END.
```

Note here that the **INTERFACE** reserved word must come before the **USES** statement, if any. Nothing, in fact, may come between the unit name and the **INTERFACE** reserved word.

One type and a variable of that type are defined in the interface section shown. Both of these definitions are "visible" to any program or unit that uses **Skeleton**. A function and a procedure are also defined in the interface part of **Skeleton**, and like **MyType** and **MyVar** are visible to any unit or program that uses **Skeleton**.

Now look down to the implementation section of the unit, where the bodies of the function and procedure are given. Notice that a variable name **PrivateVar** is declared in the implementation section. As its name implies, **PrivateVar** is known *only* within the implementation section of the unit. No other program or unit can reference the identifier PrivateVar or in any other way know that it exists or how it is defined.

PrivateVar can, however, be accessed by any of the procedures or functions defined within the unit. So, in a sense, the effects of **PrivateVar** can be "felt" by outside programs or units that use the subprograms that have access to **PrivateVar**, but the variable itself remains invisible to anything outside the unit in which it is declared.

Why is this important? Like so many of Pascal's structural limitations, it is done to minimize the possibility of undeclared "sneak paths" occurring between routines when those paths are not desired. Such sneak paths make possible bugs of a truly insidious nature.

A useful unit example

A very simple example involves the **MakeBox** procedure that I presented in Section 10.2. **BoxStuff** is the unit used by the **Rollem** dice game I presented in Section 11.1. I've gathered together the **LineRec** and **PCLineChars** definitions, along with the **MakeBox** procedure itself, and created a short unit to "box it all up" and make it all available simply by placing the name **BoxStuff** in your **USES** statement. Read it carefully to make sure you understand all the details.

```
{------------------------------------------------------------}
{                          BoxStuff                          }
{                                                            }
{ Unit to demonstrate separate compilation -- draws text boxes }
{                                                            }
{                        by Jeff Duntemann                   }
{                        FreePascal 3.0.4                    }
{                        Last update 11/15/2019              }
{                                                            }
{     From: FREEPASCAL FROM SQUARE ONE  by Jeff Duntemann    }
{------------------------------------------------------------}

UNIT BOXSTUFF;

INTERFACE

USES Crt;      { For GotoXY }
```

```
TYPE
  LineRec = RECORD
               ULCorner,
               URCorner,
               LLCorner,
               LRCorner,
               HBar,
               VBar,
               LineCross,
               TDown,
               TUp,
               TRight,
               TLeft : String[4]
             END;


{ PCLineChars: }
{ Contains box-drawing strings for MakeBox.}
{ Any program or unit that USES BoxStuff   }
{ can access the PCLineChars constant just }
{ as though it had been defined within the }
{ USEing program or unit.                  }

CONST
  PCLineChars : LineRec =
    (ULCorner : #201;
     URCorner : #187;
     LLCorner : #200;
     LRCorner : #188;
     HBar      : #205;
     VBar      : #186;
     LineCross: #206;
     TDown     : #203;
     TUp       : #202;
     TRight    : #185;
     TLeft     : #204);


{<<<< MakeBox >>>>}
{ This is all that the "outside world" really needs to see of the }
{ MakeBox procedure.  *How* it happens is irrelevant to using it. }

PROCEDURE MakeBox(X,Y,Width,Height : Integer;
                  LineChars        : LineRec);
```

```
IMPLEMENTATION

{ <<<<MakeBox>>>> }

PROCEDURE MakeBox(X,Y,Width,Height : Integer;
                 LineChars        : LineRec);

VAR
  I : Integer;

BEGIN
  IF X < 0 THEN X := (80-Width) DIV 2;    { Negative X centers box }
  WITH LineChars DO
    BEGIN                                 { Draw top line }
      GotoXY(X,Y); Write(ULCorner);
      FOR I := 3 TO Width DO Write(HBar);
      Write(URCorner);
                                          { Draw bottom line }
      GotoXY(X,(Y+Height)-1); Write(LLCorner);
      FOR I := 3 TO Width DO Write(HBar);
      Write(LRCorner);
                                          { Draw sides }
      FOR I := 1 TO Height-2 DO
        BEGIN
          GotoXY(X,Y+I); Write(VBar);
          GotoXY((X+Width)-1,Y+I); Write(VBar)
        END
    END
END;


END.
```

15.4. INITIALIZATION AND FINALIZATION

Structurally, units are very much like Pascal programs without program bodies. The important parts are the definitions of constants, types, variables, and subprograms. Now, a unit may in fact have one or both of two additional things: the unit's initialization section, and its finalization section. In FreePascal, a unit may include one, or the other, or both. This differs from Delphi, which will not compile a unit's finalization section if there is not also an initialization section. It's easy to beef up a version of our unit Skeleton to include both initialization and finalization:

```
UNIT Skeleton;

INTERFACE

IMPLEMENTATION

INITIALIZATION
  BEGIN
  END;

FINALIZATION
  BEGIN
  END
END.
```

One thing that may look odd is that the two words **INITIALIZATION** and **FINALIZATION** are in uppercase. In this book at least (and in my tutorials generally) reserved words are in all caps, and the two are reserved words in nearly all implementations of Object Pascal, including Delphi. Neither is required for a unit to compile. As you'll notice, the example unit BoxStuff presented earlier has neither.

The **Skeleton** unit as shown above will compile, even though it's empty and generates no code.

When initialization sections run

In a program that uses several units, the initialization section (if there is one) for each one of those units will execute before the main program body begins executing. The order in which the unit initialization sections run is the same order that the units are named in the **USES** statement, from left to right. For example:

```
USES Crt,Mouse,BoxStuff;
```

When the program that contains this statement is run, the initialization section for **Crt** executes first, followed by the initialization section for **Mouse**. There is no initialization section for the unit **BoxStuff**, so as soon as the initialization section of **Mouse** finishes executing, the main program body begins executing.

The finalization sections of the units used in a program will execute *in reverse order* after execution of the program itself terminates. For example, in the **USES** statement above, the units will be initialized in the order **Crt**,**Mouse**,**BoxStuff**. When the program ends, the finalization sections of the three units will be executed in the order **BoxStuff**,**Mouse**,**Crt**.

Of what use are initialization sections and finalization sections? There are several, although most of them are relatively advanced concepts that you may not need to use until you have come up to speed in Pascal programming in general and (especially) begin using objects.

Most simply, an initialization section can initialize global variables declared in the unit to some desired initial value. This relieves the program itself of the responsibility, and avoids the possibility that the programmer will forget to add initialization code to the beginning of his program that uses the unit. Also, in situations where a vendor sells a unit as a separate product, the initialization section guarantees that any globals that need to be initialized will be initialized so that the unit (which may not be sold with source, and hence not fully understood by the programmer who works with it) will work correctly without depending on the programmer. Also, a unit may need to allocate memory on the heap for data structures.

Finalization sections are used less often. Mostly they release memory that was allocated by code somewhere in the unit. Again, this will make more sense (and become more useful) once you begin using objects, pointers, and the heap. Alas, I cannot cover those topics in this one introductory volume.

What's Next?

Many people who have downloaded this PDF since 2011 have written to ask me how much it will cover once I consider it complete—and what my next project involving FreePascal/Lazarus will be. Well, with the 5/5/2025 revision, I consider this book complete. Let's talk about that a little.

First of all, remember what I'm trying to do with this book: Create an introduction to not only FreePascal but to the ideas of programming itself. It's intended to be accessible to people who have never coded before at all, in any language. This means that the first 90-odd pages of the book will be unnecessary for people who are already familiar with the general principles of programming and especially Pascal programming. People who know programming but are new to Pascal can skip past the first 60 pages or so.

I made a deliberate decision to limit this book to about 350 pages. That's a lot of paper to print, punch, and bind if you want to make a paper copy for yourself. I intend to post a spiral-bound paper copy on Amazon KDP soon, though that will obviously not be a "free" book.

In other words, I want to keep to the mission of a book not only for newcomers to Pascal, but even newcomers to programming itself.

Here's a short list of things I will *not* cover:
- Object-oriented programming (OOP)
- Traditional pointers and linked lists (they've been subsumed by OOP Lists)
- GUI building with Lazarus
- Database programming

All of that depends on objects, and Lazarus components are objects. So the next book, whatever its title, will begin with OOP and move from there to GUI building with Lazarus and then everything else. I hope to keep my future books shorter, so I may move database work off to an entirely separate book. I have no timetable at this point. They'll happen when they happen. Hang in there. Thanks for downloading, and double thanks for being interested in Pascal!

—73—

Jeff Duntemann K7JPD
Scottsdale, Arizona, USA