April 1993

Introduction to Electrical Engineering and the Art of Problem Solving: Volume II

P. Frazer Williams
University of Nebraska - Lincoln, pfw@moi.unl.edu

INTRODUCTION TO ELECTRICAL ENGINEERING AND THE ART OF PROBLEM SOLVING

Notes for EEngr 121 and 122

Volume II

©*Frazer Williams*
University of Nebraska-Lincoln
Fall, 1993

PREFACE

These notes were written to accompany a two-semester introductory course in Electrical Engineering. The primary goal of the course is to introduce you to the art of technical problem solving. Conventional Electrical Engineering curricula, in favorable cases at least, produce engineers who are reasonably adept at operating many of the tools important to the field, such as mathematics and circuit analysis. Often, however, these engineers lack skill in choosing which tools to use and in designing a plan of attack to solve problems which are new to them, even though they have all the information and skills needed to fabricate a satisfactory solution. Problem solving is more of an art than a science, and it can only be learned through experience. I hope these notes and the homework exercises will help you to develop this important ability early in your academic career.

Problem solving is not something that can be learned by just listening to some lectures or reading some material, no matter how well prepared. I don't know of any specific set of reasoning or rules I can give you to make you proficient at it. You just have to jump in and flounder around for a while. I can give you some suggestions, however, which may keep you from drowning.

1. Make sure you understand the problem. If it's a homework or a test problem, make sure you understand what all the words mean and how they fit together. Before you start looking for "the answer," make sure you understand what the problem is.
2. Spend a little time deciding on a plan of attack. As you go along check frequently to make sure that you are following your plan. If it appears that the original plan won't work, change it; but do so intentionally, not accidentally.
3. Plan on spending some "quality" time on the homework. As a guideline, you should expect to spend on the average two to three hours outside of class for every lecture hour.
4. Your goal in doing homework is to arrive at an answer through a process you create, not just to get an answer, even if it's right.
5. If, after spending some time, you can't get anything to work out on a problem, seek help from the instructor, teaching assistant, or another student. Use such help wisely. Again, the main goal is not to get an answer, but rather to see where you went wrong or what idea you missed.
6. Try to view each homework problem as an opportunity to hone your problem solving skills. Remember that there is a good chance that problems similar, but not identical, to the homework problems will be on the tests. If you were unable to solve a problem in the relaxed atmosphere of home, it is unlikely you will be able to solve its cousin in the stressed atmosphere of an exam.
7. There are often several reasonable ways to solve a problem. I suggest trying to solve homework and example problems in more than one way. This is a good way to study for tests. If you work a problem in two different ways that both seem correct to you, and get different answers, try to figure out why. If after a little time both approaches still seem correct, talk to the instructor.

In writing the notes, I've tried to find non-trivial problems for which you already know, or I can easily tell you, everything needed to obtain a solution. The difficult part is sorting the useful information from the rest of the stuff you know, and putting it all together in a way that works. The important aspect of these discussions is the *process*, not the result. If the primary goal were to obtain the answer, often the best method of doing so would be something other than that in the notes. (In cases in which the method discussed is obviously a poor choice, I've tried to point that out, along with at least some directions pointing toward the better choice.)

The notes have two secondary goals. The first is to show you some of the ideas that you will encounter a little later in your academic career, and to solidify your grasp of some ideas you have seen already. Seeing the new ideas now and giving them time to "rattle around" in your head for a while will make them easier to master when you encounter them later in more specialized courses. For example

Chapter 8 is devoted entirely to the analysis of simple circuits in order to determine currents and voltages. The purpose here is not to make you proficient at circuit analysis, but rather to introduce you to the basic ideas, such as just what is a current or a voltage. Then, when you encounter the subject again in considerably more detail, you will have some understanding and familiarity to fall back on.

The other secondary goal is to provide you with skill in using a computer. In the first semester of the course, your interaction with a computer will be almost entirely through a spreadsheet program. This is a pretty specialized program which turns out to be surprisingly useful in engineering. It allows you to graph data and functions, and to do rather complex arithmetic calculations on large quantities of data easily. I hope that you will come to view the spreadsheet as a tool you can use later to help you understand things and to avoid some of the arithmetic drudgery. This is the easiest goal of the course to achieve. I think you should find the spreadsheet easy to learn and entertaining to use (it was designed that way). In the second semester, you will be introduced to a more general-purpose program, called the C compiler. C is a computer language somewhat similar to FORTRAN, Pascal, or BASIC which allows you to write your own programs, and thereby to tell the computer exactly what you want it to do. You will probably not emerge from the course as a skilled C programmer, but I hope you will be able to use C to solve many of the problems which you are likely to encounter later in your career. Perhaps most importantly, I hope you will have a good idea about just what can be done with C, and what can't, so that you will be able to further develop your skill at using C as the need arises.

I hope you will find these notes interesting, and at least in some parts fun. The core of the material in the notes should be accessible to all students in the class, but I've also tried to include material which even the best students will find challenging. This material usually comes at the end of the discussion on the topic.

On a personal note, I consider myself to be very fortunate that part of the job for which I get paid is something I really enjoy doing. I really like figuring out how to make something, designing it, putting it together, and finally seeing it work. The "something" can be a physical thing such as an electronic circuit or a piece of mechanical equipment, or something more ephemeral like a clever solution to a mathematical problem or a computer program. The part of the process I do *not* like is the part involving doing nearly the same thing I've done many times before, such as soldering the circuit together or laying out and drilling the holes, or doing the algebra or actually typing the program into the computer. The part I *do* like is getting the idea, and then seeing it actually work. I even liked putting these notes together! I really disliked writing them and getting everything just right, put thinking up how to present the material and seeing it go together was enjoyable.

It seems to me that many students go through their college careers choosing to see only the boring, unpleasant stuff, like the mechanics of circuit analysis. You are going to have to do your share of algebra and the like, but I hope that when given the chance, you will also look around and see the fun stuff, as well. It's all over the place. In writing these notes, I've tried to show you a few of the things I've found enjoyable.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

11. FINITE SUMS AND INTEGRATION

The purpose of the next few sections is twofold. First, I want to introduce you to some ideas which are very useful for using a computer to help solve problems involving either integration or differentiation. Second, I want to reinforce some of the ideas that you may have first seen last semester in Math 106. It has been my experience that for many students these ideas are a little shaky. The ideas are widely used in electrical engineering, and it is important that you feel comfortable with them.

Our approach to the subject will be different than what you probably saw in the math course, and I hope the alternate view will help solidify the ideas. In both this chapter and the next, we'll first consider a concrete example in which an important idea from calculus can be used. I chose these examples because I thought they would be easy to visualize, and I hope that you will spend some time trying to do that. The ideas of both integral and differential calculus are rather simple, but very powerful. If you really understand the idea you will have acquired a very useful tool which you can use in a wide range of engineering fields.

To make visualization easier, we will first solve the problems numerically, using Quattro Pro. Instead of dealing with abstract ideas like limits, we will approximate integrals and derivatives by using Quattro Pro to do simple arithmetic like addition and subtraction. I hope that this approach will make it easier for you to visualize just what an integral and a derivative really is, and in the process perhaps clarify the idea behind taking a limit. For most of the problems an analytic solution exists, and we'll make use of this fact to compare the results of the numerical solution to the analytic, and to reinforce the ideas involved in obtaining the analytic solution.

11.1 How Much Water Can a Water Tower Hold?

The town of Notrees, Texas has a water tower just outside town. The tank of the tower is a sphere 20 meters in diameter, with the bottom of the tank 15 meters above the ground. The question is, how much water does the tank hold? There are several ways to answer this question. You probably know that the volume of a sphere of radius R is $V = \frac{4}{3}\pi R^3$. The volume of water that the tank can hold is, then, $\frac{4}{3}\pi \times 10^3 \approx 4189$ cubic meters. Instead of solving the problem using this formula, I'd like to do it another way—well actually two other ways. As with the problems in Chapter 10, I'll develop first a numerical solution, and then an analytic solution (that's the second way). My main interest here is showing you some different ways of looking at what needs to become a familiar concept—integration. If, on the other hand, I were only interested in solving this problem, I'd just use the formula.

11.2 Numerical Solution

In this sub-section we'll estimate the volume of the sphere numerically, using Quattro Pro. The techniques I'll discuss are applicable to and useful for other shapes besides a sphere. For this discussion I've chosen to consider a sphere because it is easily visualized, and there is a commonly-known formula for the volume of a sphere which can be used to check how well the ideas work. What I would like you to get out of the following material is *not* another recipe for determining the volume of a sphere—the known formula, $\frac{4}{3}\pi r^3$, is much more convenient and accurate. Rather, the important point is to understand the *idea* behind the technique. This idea can be applied to other shapes for which such a simple formula does not exist. Further, the idea is the same as the one behind the integral calculus, and I hope that seeing this idea applied to a concrete and easily visualized example will reinforce what you have already learned about it in Math 106. If you really understand this idea, you should find it easy to apply the integral calculus to a wide range of engineering problems, and you should find it much easier to figure out all types of integrals such as line, surface and volume integrals when you encounter them later.

Here's the plan of attack. Since I'm going to pretend I don't know a formula for the volume of a sphere, I'll try to divide the tank up into several pieces, the volumes of which I do know a formula for, and

then I'll add up the volumes of each of the pieces. There are several ways to make the division, but I choose to divide the tank into a number of thin slices as shown in Fig. 11−1a.

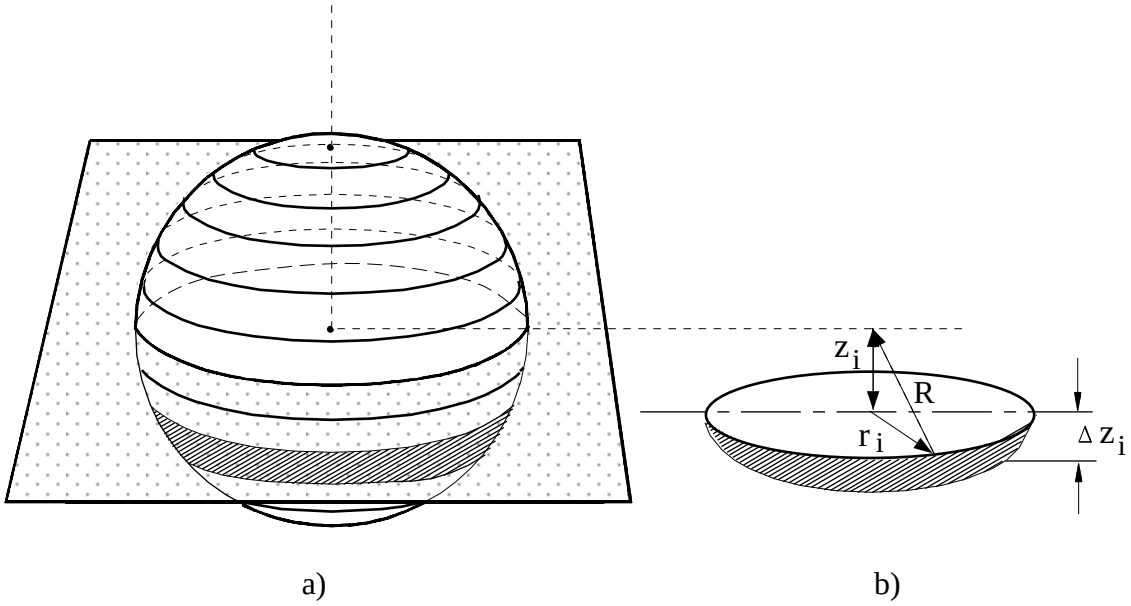

Figure 11−1. a) The division of the sphere into a number of thin slices. b) One representative slice, showing the coordinates used to estimate its volume.

What is the volume of one of these slices? Fig. 11−1b shows one of them. Except for a little detail, the answer is easy. If the edges of the disks were vertical, the volume would just be the area of the circular face times the thickness. If the volume of the i^{th} slice is ΔV_i, the radius of the face of the slice r_i, and the thickness of the slice Δz_i, then

$$\Delta V_i \approx \pi\, r_i^2\, \Delta z_i \tag{11−1}$$

This result isn't exactly right because the edges are not vertical, but I can make the error as small as I like by choosing thin enough slices. The thinner the slices, though, the more volumes I have to add together to get the total volume.

This is the main idea behind the integral calculus—to calculate the integral of some quantity over some region, divide the region up into a number of small pieces such that you can figure out approximately the integral over each piece, and then add up the integrals over each piece. Doing that will give you an approximate value for the integral. If the division is such that the error in approximating the integrals over the individual pieces decreases as the size of each piece decreases (and hence the number of such pieces increases), then you can get as accurate an answer as you want by just taking the number of pieces large enough. The integral is defined as being *exactly* the limit as the number of pieces becomes infinitely large. We cannot consider an infinite number of pieces using a spreadsheet, but we can use a number large enough to get a pretty good approximation. We will do just that with this water tower example, and see if the approximate answer we get does in fact approach the exact result at the number of slices increases.

Let me say a word about notation here that may save you some confusion. The symbol ΔV_i is one quantity, *not* the product of two quantities, Δ and V_i. This kind of notation is frequently used to denote small quantities. I used it in Eq. (11−1) to imply that the thickness of the slice, Δz_i, is supposed to be small, and that the volume of the slice, ΔV_i, will than also be small. The radius of the face of the slice, r_i, is not generally expected to be small, so I did not use the Δ notation with it.

You may be thinking that my plan for calculating the volume is a little shaky. It is true that the error in Eq. (11−1) decreases if we decrease the slice thickness, Δz_i, but it is not so obvious that the error in estimating the volume of the sphere decreases as well, because we must add up more ΔV's. Each individual error is smaller but we make more errors, and it is not clear that the overall result will be an improvement. In fact, the overall error does decrease with decreasing slice thickness. There are two ways to show this. The first is the way we'll do here, and is a kind of an experimental approach. We'll simply program Quattro Pro to add up the volumes as per the plan, and then see if the answer approaches some fixed value as we decrease the slice thickness. If it does, great; if it doesn't, we have to go back to the drawing board.

The second way is an analytic one in which one estimates the error in taking the volume of a slice to be given by Eq. (11−1). The error is just the volume of a ring with a roughly triangle-shaped cross-section. I don't want to go into it here, but it can be shown that this error is proportional to Δz_i^2. The total number of such errors we make in adding up the total volume is just the number of slices. Assuming that each slice has the same thickness, this number is proportional to $1/\Delta z$, so the total error is proportional to the product, $\Delta z^2 \, \frac{1}{\Delta z} = \Delta z$. Thus as Δz is made smaller, the overall error should get smaller as well. In fact, the error should be roughly proportional to Δz.

Back to the problem, we have divided the sphere into a number of slices, and we have an approximate expression for the volume of each. We need only to program Quattro Pro to evaluate the volume of each slice and add up each individual volume for us. The top and bottom halves of the sphere are the same, so I'll estimate only the volume of the bottom hemisphere, and then double the result to get the volume of the whole sphere. To estimate the volume of each slice using Eq. (11−1), I need a formula for the radius of each slice, r_i. We know that all points on the surface of a sphere are the same distance from the center, and that distance is the radius of the sphere, say R. Then as in Fig. 11−1b if z_i is the distance of the top of the i^{th} slice from the center of the hemisphere, the radius of the top face of the slice is (from the Pythagorean theorem)

$$r_i^2 + z_i^2 = R^2$$

or

$$r_i^2 = R^2 - z_i^2 \tag{11−2}$$

Eq. (11−2) gives the radius of the top face of the i^{th} slice in terms of the distance of this slice from the top of the hemisphere, z_i. To figure out what z_i is, we have to decide on how we'll slice up the hemisphere. Let's choose to divide the hemisphere into N slices, each of the same thickness. Then $\Delta z_i = \Delta z = R/N$, and

$$z_i = (i-1)\,\Delta z \tag{11−3}$$

This result can be used in Eq. (11−2) to calculate r_i^2, and that result then can be put in Eq. (11−1) to estimate ΔV_i. All that then remains is just to add up all N ΔV's, and multiply by 2 to get the volume of the sphere.

Let's put it all into the worksheet. I'll tell you step-by-step how I set my worksheet up. I suggest you design your own first, and then compare with mine. The way I chose is certainly not the only way, and it may not even be the best way. I put the slices each in a different row, starting with the top slice. I used column `A` for the slice number, i, column `B` for the distance of the top of the slice from the top of the hemisphere, z_i, column `C` for the square of the radius of the top face of the slice, r_i^2, and `D` for the volume of the slice, ΔV_i. I started with slice 1 in row `10` to let me put a title and the values of the parameters and the answer at the top of the worksheet. I used cell `A7` for the radius of the sphere (10 in this case), cell `B7` for the number of slices, and cell `C7` for Δz. In cell `E7` I put the numerical result for the volume of the sphere, and in cell `F7` the known answer, $\frac{4}{3}\,\pi\,R^3$.

For the formulae, in cell `A10` I put `+A9+1`, and in `A11` I put `+A10+1`. In `B10` I put

`(+A10-1)*$C$7`, in `C10` I put `$A$7*$A$7-B10*B10`, and in `D10` I put `@PI*C10*$C$7`. In cell `A7` I put `10` (for the radius of the sphere), in cell `B7` I put `@COUNT(A10..A500)` (for the number of slices), and in cell `C7` I put `+A7/B7` (for Δz). In cell `E7` I put the total calculated volume, `2*@SUM(D10..D500)`, in cell `F7` I put `4*@PI*A7^3/3`. Finally, I prettied it up by putting in labels and some lines. Fig. 11−2 shows the worksheet.

a)

	A	B	C	D	E	F
1	Chapter 10: Numerical calculation of the volume of					
2	a sphere.					
3						
4						
5	PARAMS	CONSTANTS			VOLUME	
6	R	N	delta z		Numeric	Analytic
7	10	@COUNT(A10..A500)	+A7/B7		2*@SUM(D10..D500)	4*@PI*A7^3/3
8						
9	Slice	z	r^2	Volume		
10	+A9+1	(+A10-1)*$C$7	+$A$7*$A$7-B10*B10	@PI*C10*$C$7		
11	+A10+1	(+A11-1)*$C$7	+$A$7*$A$7-B11*B11	@PI*C11*$C$7		

b)

	A	B	C	D	E	F
1	Chapter 10: Numerical calculation of the volume of					
2	a sphere.					
3						
4						
5	PARAMS	CONSTANTS			VOLUME	
6	R	N	delta z		Numeric	Analytic
7	10	40	0.25		4266.67552	4188.7902
8						
9	Slice	z	r^2	Volume		
10	1	0	100	78.53982		
11	2	0.25	99.9375	78.49073		
12	3	0.5	99.75	78.34347		
13	4	0.75	99.4375	78.09803		
14	5	1	99	77.75442		
15	6	1.25	98.4375	77.31263		
16	7	1.5	97.75	76.77267		

Figure 11−2. The first few rows of the worksheet I programmed to calculate the volume of the spherical water tank. Part a) shows the formulae in each cell, and part b) shows the appearance of the worksheet on the screen.

Now, to use the worksheet we need only decide how many slices to divide the hemisphere into, and then copy `A10..D10` down that many times. For example, I started by dividing the sphere into 10 slices,

so I copied `A10..D10` into the range `A11..D19`. Doing that gave me a so-so result, $V \approx 4492$. Since the exact result is 4189, and the error is then 304, or a little less than 10%. Table 11−1 gives the results obtained for several different numbers of slices.

1st ORDER METHOD			
Number of Slices	**Δz**	**Approx. Volume**	**Error**
10	1.0	4492	304
20	0.5	4343	154
40	0.25	4267	77.9
100	0.10	4220	31.3

TABLE 11−1. Results of the program as a function of the number of slices used.

A few paragraphs back, I discussed the concern that the method might not work at all, because it was not obvious that the overall error would decrease as the number of slices is increased. From Table 11−1 it is clear that the error does decrease as the number of slices increases (or Δz decreases), so we have shown "experimentally" that the method is valid, at least for spheres. In fact, the error seems to be very nearly proportional to Δz. For example, in going from $\Delta z = 1.0$ to 0.25, a factor of 4, the error decreases from 304 to 77.9, a factor of 3.9. Although I did not derive it for you I mentioned that it is possible to show analytically that the error should be roughly proportional to the slice thickness, Δz. The "experimental" results seem to bear this prediction out. Such a method in which the error decreases linearly with the "slice" thickness is said to be *first order* because the error depends on the first power of Δz.

Since we always take the volume of a slice to be the volume of a disk of radius equal to the radius of the *upper* face of the slice, we will each time over-estimate the volume. Thus we would expect our result to be too big, as found. If we had used the radius of the *lower* face of each slice to estimate its volume, we would have found similar behavior, except that we would have always under-estimated the volume, and would have gotten a result that was a little too small.

This leads to what turns out to be a good idea: suppose we use the radius of the slice at a point halfway between the top and bottom faces to estimate the volume of the slice. The estimation still won't be exact, but it should be a lot closer. Making this change in our worksheet is easy. As programmed, the worksheet uses the distance from the top of the hemisphere to the top of the slice to determine the radius to be used in calculating the volume of the slice. To change to the new idea, we need only use instead the distance from the top of the hemisphere to the center of the slice. That corresponds to just adding $\frac{1}{2}\Delta z$ to the value used in column `B`. The formula used was $z_i = (i-1)\Delta z$. This needs to be changed to

$$z_i = (i-1)\Delta z + \tfrac{1}{2}\Delta z = (i-\tfrac{1}{2})\Delta z \qquad (11\text{−}3')$$

Before making the change, I suggest saving the original worksheet to a file for later use. Then you can simply use the `EDIT` key (`F2`) to change the formula in `B10` to `(+A10-0.5)*$C$7`. This change can then be propagated downward with the `COPY` item of the `EDIT` menu.

The change with this improvement is striking. Table 11−2 shows the results I obtained. Two things stand out from these results. First, the error obtained with only 10 slices with the new method is less than the error obtained with 100 slices with the old, so this really was a good idea! Second, the error is approximately proportional to the *square* of Δz. (For example, the error with 20 slices is exactly a quarter of that with 10 slices, and the error with 100 slices is 0.01 times that with 10.) It is somewhat harder, but it can be

2nd ORDER METHOD			
Number of Slices	Δz	**Approx. Volume**	**Error**
10	1.0	4194	5.24
20	0.5	4190	1.31
40	0.25	4189	0.327
100	0.10	4189	0.0524

TABLE 11−2. Results of the new, improved program as a function of the number of slices used.

shown that the error in estimating the volume of a slice with this method is proportional to Δz^3 rather than Δz^2 as with the previous worksheet. Multiplying the error per slice by the number of slices, $\frac{R}{\Delta z}$, gives an overall error proportional to Δz^2. A method such as this is said to be of *second order*.

Let me reiterate a point I made at the start of this section. If your only interest were simply to find out the volume of the sphere, it would be dumb to calculate the volume this way. Much better would be to use the known formula, $V = \frac{4}{3}\pi R^3$. The method I've just discussed, however, could be used for other shapes of water tanks for which the formula for the volume is not known. For example, if the water tank were egg-shaped this technique could be used. The only tricky point would be figuring out a formula analogous to Eq. (11−2) for the square of the radius of the i^{th} slice.

11.3 Analytic Solution

There exists an analytic solution to the problem of the volume of a sphere. (That's where the formula for the volume of the sphere came from.) We have almost done all the work to figure it out; we only have to put it all together. Using Eq. (11−2) for r_i^2, Eq. (11−3) for z_i, and $\Delta z_i = \frac{R}{N}$ in Eq. (11−1), we obtain

$$\begin{aligned} \Delta V_i &\approx \pi \left[R^2 - (i-1)^2 \left(\frac{R}{N} \right)^2 \right] \frac{R}{N} \\ &= \frac{\pi R^3}{N} \left(1 - \frac{(i-1)^2}{N^2} \right) \end{aligned} \tag{11−4}$$

The total volume of the hemisphere is then

$$\begin{aligned} V_{hemi} &\approx \sum_{i=1}^{N} \Delta V_i \\ &= \frac{\pi R^3}{N} \sum_{i=1}^{N} \left(1 - \frac{(i-1)^2}{N^2} \right) \\ &= \frac{\pi R^3}{N} \left(\sum_{i=1}^{N} 1 - \frac{1}{N^2} \sum_{i=1}^{N} (i-1)^2 \right) \\ &= \frac{\pi R^3}{N} \left(N - \frac{1}{N^2} \sum_{i=1}^{N} (i-1)^2 \right) \end{aligned} \tag{11−5}$$

where in the last equation we have used $\sum_{i=1}^{N} 1 = N$.

We still need to evaluate the remaining sum in Eq. (11−5). We can make one simplification pretty easily. Notice that in the sum i goes from 1 to N, whereas the quantity $(i-1)$ inside the summation sign goes from 0 to $N-1$. Thus we can write

$$\sum_{i=1}^{N} (i-1)^2 = \sum_{j=0}^{N-1} j^2 \tag{11−6}$$

(If you don't see that, just write out the first few terms of each sum.) It turns out that there is a formula giving the value of the sum in Eq. (11−6) as a function of N. For now, I'll just tell you what it is, but if you are curious, I've put a derivation in an appendix to this chapter. The formula is *not* immediately obvious.

$$\sum_{j=0}^{N-1} j^2 = \tfrac{1}{6}\, N(N-1)(2N-1) \tag{11−7}$$

I suggest you check it for a few small values of N.

With this result, we can obtain an analytic formula for the volume of the hemisphere.

$$\begin{aligned} V_{hemi} &\approx \frac{\pi R^3}{N}\left(N - \frac{1}{N^2}\,\frac{1}{6}\, N(N-1)(2N-1)\right) \\ &= \pi R^3\left(1 - \frac{1}{6}\,\frac{(N-1)(2N-1)}{N^2}\right) \end{aligned} \tag{11−8}$$

This is the volume of the hemisphere, so to get the total sphere volume, you have to multiply it by two. You might find it interesting to put this formula into your worksheet to check it against what you get by actually summing up the volumes. If you do, remember that this formula was obtained with $z_i = (i-1)\Delta z$, which means that the radius used in estimating the volume of the slice was that of the upper face, not of the middle, so to compare you should use the first worksheet we developed, not the second.

Now, the result in Eq. (11−8) only approximately gives the volume of the hemisphere for any finite value of N because the formula we used for the volume of a slice was not exactly right. (Eq. (11−8) had better be approximate because it claims that the volume of the sphere depends on the number of slices, and that clearly cannot be true. Slicing up the sphere is something I only do in my mind and unless I'm psychokinetic the volume of the sphere can't depend on how I decide to slice it up.) We can make the result as accurate as we please by choosing N sufficiently large. What happens to Eq. (11−8) if N is really big, like a zillion? A zillion minus 1 is essentially the same as a zillion, and two zillion minus 1 the same as two zillion, so

$$\frac{(N-1)(2N-1)}{N^2} \underset{N\to\infty}{\longrightarrow} \frac{(N)(2N)}{N^2} = 2$$

Finally, the exact result for V_{hemi} is

$$V_{hemi} = \pi\, R^3 (1 - \tfrac{2}{6}) = \tfrac{2}{3}\,\pi\, R^3 \tag{11−9}$$

Multiplying this result by two to get the volume of a full sphere gives the famous result.

11.4 Just How Much Water Does the Notrees Tank Hold, Anyway?

The material in this section is a short side-track which has little to do with any of the other sections in this chapter. We have shown by several methods that the water tank in Notrees can hold 4189 cubic meters of water. In this section, I'd like to consider just how much water this is. In engineering it is important that you have the best intuitive feel you can for the magnitudes of the quantities involved in any design project. Without such a feel, you may end up designing something that cannot be built, or you may make an error

which makes the result of some calculation obviously ridiculous and not realize it. In this section I'll not be interested in high accuracy, but rather in converting 4189 cubic meters into some measure I have a better feeling for.

Let's first estimate how much the water weighs. The density of water is 1 g per cubic centimeter. (That's a good number to remember. Most solids and liquids have about the same density. I've also found "A pint's a pound the world around" useful.) One cubic meter contains $100^3 = 10^6$ cubic centimeters, so a cubic meter of water weighs 10^6 g $= 1000kg$. Wow! That's more than a ton! Anyway, the tank can hold $4189 \times 1000 \approx 4.2 \times 10^6$ kg. Perhaps you would like this in gallons. I don't remember how many cubic meters there are in a gallon, but I do remember that 1 kg is about 2.2 lb, and with the bit of poetry above, I remember that a lb is a pt. Therefore that tank can hold $4.2 \times 10^6 \times 2.2 \approx 9.2 \times 10^6$ pints. There are 8 pints in a gallon, so the tank can hold $9.2/8 \approx$ one million gallons. That sure sounds like more water than does 4189 cubic meters. If the population of Notrees is 1000, how long would this water last them? Say the average person uses 100 gallons of water a day. Then Notrees uses $1000 \times 100 = 10^5$ gallons per day, and the tank would last them about 10 days.

11.5 Connection with Integrals

In calculating the volume of the water tank, we have actually been evaluating an integral. To see that, remember that

$$V_{hemi} \approx \sum_{i=1}^{N} \Delta V_i$$

Then using Eqs. (11−1) and (11−2),

$$V_{hemi} \approx \pi \sum_{i=1}^{N} (R^2 - z_i^2)\, \Delta z_i \tag{11−10}$$

This expression becomes exact in the limit that we take an infinite number of slices,

$$V_{hemi} = \pi \lim_{N \to \infty} \sum_{i=1}^{N} (R^2 - z_i^2)\, \Delta z_i \tag{11−11}$$

Since Δz_i is the thickness of each slice, it must become very small as N becomes very big. The limit in Eq. (11−11) is just the definition of the integral of $(R^2 - z^2)\, dz$ from $z = 0$ to $z = R$,

$$V_{hemi} = \pi \int_0^R (R^2 - z^2)\, dz \tag{11−12}$$

The advantage of writing the volume this way is that someone has already figured out the integral of many functions, and if we can reduce our problem to a set of integrals which are known, we don't have to mess with summations such as Eq. (11−7). In our case, we can divide the integral in Eq. (11−12) up into the sum of two integrals, both of which we know.

$$V_{hemi} = \pi \left(\int_0^R R^2\, dz - \int_0^R z^2\, dz \right) \tag{11−13}$$

The first integral is just the integral of a constant and is easy,

$$\int_0^R R^2\,dz = R^2\int_0^R 1\,dz = R^2\,z\Big]_0^R = R^2\,(R-0) = R^3 \tag{11−14a}$$

The second is only a little harder,

$$\int_0^R z^2\,dz = \tfrac{1}{3}\,z^3\Big]_0^R = \tfrac{1}{3}(R^3-0^3) = \tfrac{1}{3}\,R^3 \tag{11−14b}$$

Putting this all together, Eq. (11−13) is

$$V_{hemi} = \pi\,(R^3 - \tfrac{1}{3}\,R^3) = \tfrac{2}{3}\,\pi\,R^3$$

The volume of the whole sphere is just twice this result.

11.6 Numerical Evaluation of Integrals

The same ideas used to estimate the volume of the water tank in Section 11−2 can be used to evaluate integrals numerically. If you have an integral to evaluate, the best choice is to try to find the integral in some table of integrals, or to convert it into one or more integrals which are in your table. Failing that, you might want to evaluate the integral numerically. I'll first discuss some ideas in general, and then apply them to a specific integral so you can see how they work.

Suppose we wish to integrate some function, $f(x)$, from a to b. The procedure is similar to that we used to estimate the volume of the sphere. Instead of integrating over the whole range in one shot, we divide the range up into a number of smaller ranges, and add the integrals over each of these smaller ranges. To simplify the notation, let $x_1 = a$ and $x_{N+1} = b$.

$$\begin{aligned}\int_a^b f(x)\,dx = \int_{x_1}^{x_{N+1}} f(x)\,dx &= \int_{x_1}^{x_2} f(x)\,dx + \int_{x_2}^{x_3} f(x)\,dx + \cdots + \int_{x_N}^{x_{N+1}} f(x)\,dx \\ &= \sum_{i=1}^{N}\int_{x_i}^{x_{i+1}} f(x)\,dx\end{aligned} \tag{11−15}$$

At this point you may be wondering, how this idea helps. If I don't know how to integrate $f(x)$ over the entire range, how do I know how to do the integral over the sub ranges? The answer is that I don't, but if the sub-interval is small enough, I can approximate it the same way that I approximated the volume of each of the slices of the sphere in Section 11−2. (This is the famous ***Main Idea of Integral Calculus*** again.) Consider the i^{th} sub interval. If the width of this interval is small, $f(x)$ won't change much in crossing it, and the integral is approximately this value of f times the width of the interval, Δx_i. The integral over the entire interval is approximately just the sum of these integrals over the sub-intervals.

One contentious problem is that of what value to use for f in the i^{th} interval. In this interval x ranges from x_i to x_{i+1}, so the first try might be to use either $f(x_i)$ or $f(x_{i+1})$. In these cases, we are approximating the value of f throughout the interval by its value at one end or the other. This idea works, but not very well. I'll call it *Euler's method*. Certainly better would be to use some kind of an average value of f in the

interval. The idea here is similar to that used in developing the second worksheet in Section 11−2. There are two likely choices for the value of f to use. The first produces what is called the *rectangular rule*, and uses the value of f at the center of the interval, $f(x_i^*)$, where x_i^* is the x coordinate at the center of the i^{th} cell, $x_i^* = \frac{1}{2}(x_i + x_{i+1})$. The second produces the *trapezoidal rule*, and uses the average of f at each of the endpoint, $\frac{1}{2}[f(x_i) + f(x_{i+1})]$. Just as we found in Section 11−2 when approximating the volume of the slices of the sphere, using the value of f at either endpoint results in a first order method, (the error scales as Δx) and using the value of f somewhere near the middle results in a second order method (the error scales as Δx^2). Both the rectangular and the trapezoidal rules produce second order methods. The trapezoidal rule is the more commonly used method of the two.

We can also look at the situation graphically. If we graph $f(x)$, then $\int_a^b f(x)\,dx$ is equal to the area between the curve $f(x)$ and the x axis, for values of x ranging from a to b. Dividing the integral from a to b into a sum of integrals over sub-intervals as we did in Eq. (11−15) is analogous to dividing the area under the curve into a number of small slices as shown in Fig. 11−3.

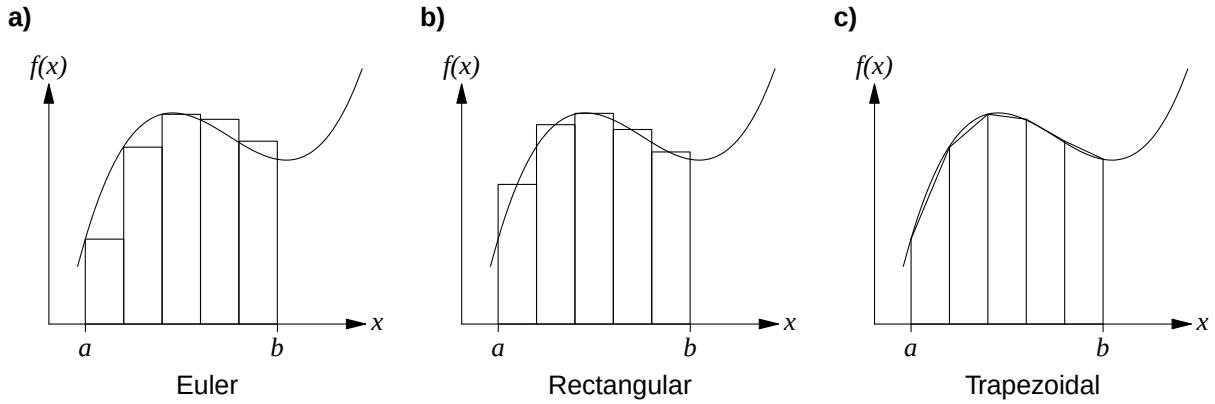

Figure 11−3. Graphical representation of a) Euler's method, b) the rectangular rule, and c) the trapezoidal rule.

Euler's method is equivalent to approximating the area under the curve by the sum of the areas of the rectangles as shown in Fig. 11−3a. The rectangular rule is equivalent to adding up the areas of the rectangles shown in Fig. 11−3b, and the trapezoidal rule is equivalent to adding the areas of the trapezoids in Fig. 11−3c.

Implementing Euler's method on a worksheet is quite straightforward. First, we divide the interval from a to b up into N sub-intervals. Although its not required, we usually choose the width of each of these sub-intervals to be the same, so

$$\Delta x_i = \Delta x = \frac{(b-a)}{N} \tag{11−16a}$$

and

$$x_i = a + (i-1)\,\Delta x \tag{11−16b}$$

Euler's method then is

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{N} f(x_i)\,\Delta x_i$$

$$= \frac{(b-a)}{N}\sum_{i=1}^{N} f(x_i)$$

where the second line only follows if we choose to make the widths of all the sub-intervals the same.

Implementing the trapezoidal rule is almost as direct. If we assume equal width sub-intervals there is one simplification that you might miss though. As for Euler's method, I'll write the approximation in a form valid even for non-equal width sub-intervals on the first line, and then I'll put the equal width result on the following lines.

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{N} \tfrac{1}{2}[\,f(x_i) + f(x_{i+1})]\Delta x_i$$

$$= \tfrac{1}{2}[\,f(x_1) + f(x_2)] + \tfrac{1}{2}[\,f(x_2) + f(x_3)] + \cdots$$
$$+ \tfrac{1}{2}[\,f(x_{N-1}) + f(x_N)] + \tfrac{1}{2}[\,f(x_N) + f(x_{N+1})]\,\frac{(b-a)}{N}$$

$$= \frac{(b-a)}{N}[\tfrac{1}{2}\,f(x_1) + \tfrac{1}{2}\,f(x_2) + \tfrac{1}{2}\,f(x_2) + \tfrac{1}{2}\,f(x_3) + \tfrac{1}{2}\,f(x_3) + \cdots$$
$$+ \tfrac{1}{2}\,f(x_{N-1}) + \tfrac{1}{2}\,f(x_{N-1}) + \tfrac{1}{2}\,f(x_N) + \tfrac{1}{2}\,f(x_N) + \tfrac{1}{2}\,f(x_{N+1})]$$

$$= \frac{(b-a)}{N}[\tfrac{1}{2}\,f(x_1) + f(x_2) + f(x_3) + \cdots + f(x_{N-1}) + f(x_N) + \tfrac{1}{2}\,f(x_{N+1})]$$

$$= \frac{(b-a)}{N}\left(\tfrac{1}{2}[\,f(x_1) + f(x_{N+1})] + \sum_{i=2}^{N} f(x_i)\right)$$

To illustrate these ideas, I'll program Quattro Pro to integrate numerically the function $f(x) = x^4$ from 1 to 2. If my only purpose were to figure out this particular integral, this would be a truly dumb way to do it. Much better would be to remember (or look up in a table) the integral of x^4,

$$\int_1^2 x^4\,dx = \tfrac{1}{5}\,x^5\Big]_1^2 = \tfrac{1}{5}(2^5 - 1^5) = \frac{31}{5} = 6.2$$

That is not my real purpose here, however. Numerical integration methods can be used to evaluate integrals which do not appear in tables, and I want to illustrate how they work. I have chosen a function with a simple, known integral so that it's easy to compare the numerical result with the correct answer.

Let's now implement Euler's method and the trapezoidal rule in a worksheet. As in Section 11−2, I'll tell you how I did it, but I suggest you experiment on your own. First I'll implement Euler's method. In column `A` I put the values of x_i, and in column `B` I put the corresponding values of $f(x_i)$. I started in row `11` to give room for parameters, constants, labels, and a title. In cell `C7` I put the number of sub-intervals, N, (I started with 100), in `A7` the value of a, 1 in this case, and in `B7` the value of b, 2 in this case. The quantity Δx is calculated from these parameters, and should not be independently set, so I put it in a cell to the side—in `E7` I put `(+B7-A7)/C7`. To generate the table of x vs. $f(x)$, in cell `A11` I put `+A7`, and in `B11` I put `+A11^4`. In `A12` I put `+A11+$E$7`, and I copied the contents of `B11` into `B12`. I then copied the range `A12..B12` down as far as I thought I would want to go, `A13..B110`.

Finally, in `D11` I evaluated the sum in Euler's method. For $N = 100$ the correct entry is `+E7*@SUM(B11.B110)`. In `E11` I put the actual value of the integral, `0.2*(B7^5-A7^5)`, and in

`F11` the error, `+E11-D11`. Fig. 11−4 shows the worksheet.

a)

	A	B	C	D	E	F
1	Chapter 10: Calculation of the integral of					
2	x^4 from a to b using Euler's method					
3						
4						
5	PARAMETERS				CONSTANT	
6	a	b	N		dx	
7	1	2	100		(B7-A7)/C7	
8						
9					INTEGRAL	
10	x	f(x)			Euler	Analytic	Error
11	+A7	+A11^4			+E7*@SUM(B11..B110)	0.2*(B7^5-A7^5)	+E11-D11
12	+A11+$E$7	+A12^4					

b)

	A	B	C	D	E	F
1	Chapter 10: Calculation of the integral of					
2	x^4 from a to b using Euler's method					
3						
4						
5	PARAMETERS				CONSTANT	
6	a	b	N		dx	
7	1	2	100		0.01	
8						
9					INTEGRAL	
10	x	f(x)			Euler	Analytic	Error
11	1	1			6.125233	6.2	0.074767
12	1.01	1.040604					
13	1.02	1.082432					
14	1.03	1.125509					
15	1.04	1.169859					

Figure 11−4. The first few rows of the worksheet I programmed to calculate the integral $\int_1^2 x^4\,dx$. Part a) shows the formulae in each cell, and part b) shows the appearance of the worksheet on the screen.

Doing all that gave me the numerical result 6.13 for the integral, and an error of 0.0748. I investigated how the error depended on the number of steps (or equivalently the step size) by carrying out the calculation for four different values of *N* ranging between 10 and 100. To change the number of steps you need to change the value in cell `C7`, and you need to edit the formula in `D11` to sum the right number of terms from the

table in column B. The results I got are shown in Table 11−3.

EULER'S METHOD		
	Numerical	
N	Result	Error
10	5.47	0.727
20	5.83	0.369
40	6.01	0.186
100	6.13	0.0748

TABLE 11−3. Results for $\int_1^2 x^4\,dx$ using Euler's method.

There are two things to notice about the results. First, the method works, but not very well. For 100 points, the error is still a little more than 1% of the value of the integral. Second, the error scales nearly linearly with N. Going from 20 to 40, a factor of 2, the error decreases by a factor of 1.98. Going from 10 to 100, a factor of 10, the error decreases by a factor of 9.72. This behavior is expected for a first order method such as this one.

Changing the worksheet to implement the trapezoidal rule is pretty easy. I suggest first saving the Euler's method worksheet for later reference. The only change required is the formula in `D11`. For the Trapezoidal rule and $N = 10$, this formula should be `+E7*(0.5*(B11+B21)+@SUM(B12..B20))`. As with the Euler's worksheet, changing values of N requires changing the contents of cell `C7`, and editing the formula in `D11` to add up the right number of elements. I evaluated the integral for the same set of N values as in Table 11−3, and the results are shown in Table 11−4.

TRAPEZOIDAL RULE		
	Numerical	
N	Result	Error
10	6.22	-0.0233
20	6.21	-0.00583
40	6.20	-0.00146
100	6.20	-0.00023

TABLE 11−4. Results for $\int_1^2 x^4\,dx$ using the trapezoidal rule.

There are two points to make from these results. First, the trapezoidal rule works a lot better than Euler's method. The trapezoidal result for $N = 10$ is better than that of Euler for $N = 100$. Second, the error scales as the square of N. As N goes from 20 to 40, a factor of 2, the error decreases by a factor of 3.99, and for N going from 10 to 100, a factor of 10, the error changes by almost exactly a factor of $10^2 = 100$. That behavior is expected from a second order method such as the trapezoidal rule.

11.7 APPENDIX: Derivation of Formula for $\sum_{j=0}^{N-1} j^2$

I know of two ways to derive Eq. (11−7). I'll describe one in some detail here and outline the other. First, to save some writing, I'll define

$$S_2(N) = \sum_{j=0}^{N-1} j^2 \tag{11−18}$$

The subscript 2 on S refers to the power of j in the sum. I'll guess that $S_2(N)$ is given by a cubic polynomial in N. Where did that guess come from? First, it truly is a guess, I don't see any obvious reason why the sum has to be given by a polynomial of any order. If my guess is incorrect, then my effort to find such a polynomial should fail. Second, why cubic? That one is a little more evident. The sum will be some kind of an average value of j^2 multiplied by the number of terms, $N-1$. The value of j runs between 0 and $N-1$, so the average value of j must be proportional to $N-1$, and the average of j^2 must scale with N something like $(N-1)^2$. Thus the sum should scale something like $(N-1)^3$, which is a cubic polynomial.

Anyway, I'll guess

$$S_2(N) = aN^3 + bN^2 + cN + d \tag{11−19}$$

where a, b, c, and d are constants to be determined, and I'll try to generate four equations which these constants must satisfy. If my guess was wrong, and the summation is not given by a cubic polynomial, I should find that the four unknowns must satisfy more than four equations. If the sum is a lower order polynomial, like a quadratic for example, I should find that I have more variables than equations.

In order to determine the values of a, b, c, and d, I'll make use of the fact that

$$S_2(N+1) = S_2(N) + N^2 \tag{11−20}$$

Substituting Eq. (11−19) into (11−20) yields

$$a(N+1)^3 + b(N+1)^2 + c(N+1) + d = aN^3 + bN^2 + cN + d + N^2$$

$$a(N^3 + 3N^2 + 3N + 1) + b(N^2 + 2N + 1) + c(N+1) + d = aN^3 + bN^2 + cN + d + N^2$$

Simplifying, and collecting terms containing the same power of N gives

$$(3a-1)N^2 + (3a+2b)N + (a+b+c) = 0 \tag{11−22}$$

The left-hand side of Eq. (11−22) must sum to zero for all possible values of N. The only way that can be true is that all the coefficients of powers of N must be individually zero.

$$3a - 1 = 0$$

$$3a + 2b = 0$$

$$a + b + c = 0$$

Solving Eqs. (11−23) is easy, giving

$$a = \tfrac{1}{3}$$

$$b = -\tfrac{1}{2} \tag{11−24c}$$

$$c = \tfrac{1}{6}$$

We have determined all the unknowns except d. We can get an equation for d, by requiring Eq. (11−19) to give the correct value of the sum for some specific value of N. I'll choose $N = 1$. Then

$$S_2(1) = \sum_{j=0}^{0} j^2 = 0 = a + b + c + d = \tfrac{1}{3} - \tfrac{1}{2} + \tfrac{1}{6} + d$$

The fractions above just add to 0, so

$$d = 0 \tag{11−24d}$$

Therefore, Eq. (11−19) becomes

$$\begin{aligned} S_2(N) &= \tfrac{1}{3} N^3 - \tfrac{1}{2} N^2 + \tfrac{1}{6} N \\ &= \tfrac{1}{6} N (2N^2 - 3N + 1) \\ &= \tfrac{1}{6} N (N-1)(2N-1) \end{aligned}$$

That is exactly Eq. (11−7).

In case you are curious, I'll outline the second way I know of for evaluating the sum in Eq. (11−7). This is a considerably more elegant method and it does not require the guess that the answer is a cubic polynomial, but it requires two very clever tricks. First, consider sums of the form

$$S_1(N) = \sum_{i=1}^{N-1} i \tag{11−25}$$

Note this is not the same as the sum we are interested in because it is the sum of the first powers of the integers, rather than the squares. This can be evaluated with one clever trick—notice that

$$\sum_{i=1}^{N-1} i = \sum_{i=1}^{N-1} (N - i) \tag{11−26}$$

The sum on the right is just the sum on the left, except that the terms are summed in the opposite order. Write out the first and last few terms to see that. Expanding the sum on the right,

$$S_1(N) = \sum_{i=1}^{N-1} N - S_1(N)$$

or

$$2S_1(N) = \sum_{i=1}^{N-1} N = N \sum_{i=1}^{N-1} 1 = N (N-1)$$

or

$$S_1(N) = \tfrac{1}{2} N (N-1) \tag{11−27}$$

This trick works for sums of any odd power of i, but unfortunately it doesn't work for even powers, such as $S_2(N)$. Instead, a second even more clever trick is required. We use the first trick to evaluate $S_3(N)$,

$$\begin{aligned} S_3(N) = \sum_{i=1}^{N-1} i^3 &= \sum_{i=1}^{N-1} (N - i)^3 \\ &= \sum_{i=1}^{N-1} (N^3 - 3N^2 i + 3N i^2 - i^3) \\ &= N^3 S_0(N) - 3N^2 S_1(N) + 3N S_2(N) - S_3(N) \end{aligned}$$

or, solving for S_3 and using the known results for $S_0(N)$ and $S_1(N)$,

$$S_3(N) = \tfrac{3}{2} N S_2(N) - \tfrac{1}{4} N^3(N-1) \tag{11−28}$$

Now comes the second clever trick; as in Eq. (11−20) we require

$$S_3(N+1) = S_3(N) + N^3 \tag{11−29a}$$

and

$$S_2(N+1) = S_2(N) + N^2 \tag{11−29b}$$

Using Eq. (11−29b) in Eq. (11−28) gives

$$\begin{aligned} S_3(N+1) &= \tfrac{3}{2}(N+1) S_2(N+1) - \tfrac{1}{4}(N+1)^3 N \\ &= \tfrac{3}{2}(N+1)[S_2(N) + N^2] - \tfrac{1}{4} N (N+1)^3 \end{aligned} \tag{11−30a}$$

Using Eq. (11−29a) and Eq. (11−28), on the other hand gives

$$\begin{aligned} S_3(N+1) &= S_3(N) + N^3 \\ &= \tfrac{3}{2} N S_2(N) - \tfrac{1}{4} N^3(N-1) + N^3 \end{aligned} \tag{11−30b}$$

Equating the right hand sides of Eqs. (11−30a) and (11−30b) gives Eq. (11−7).

11.8 Exercises

1. Consider a pyramid with base dimensions $a \times b$, and height c, as shown in Fig. 11−5.

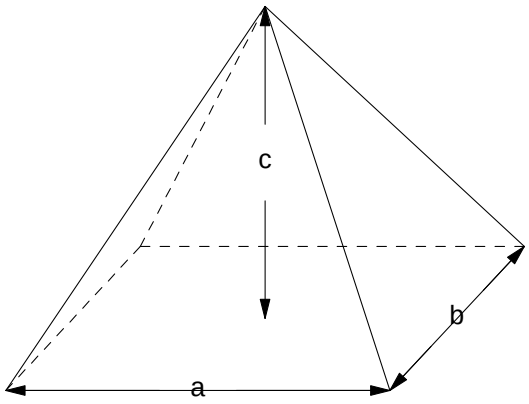

Figure 5. Drawing showing the pyramid for Exercise 11−1.

   a. Obtain a formula for the volume of the pyramid by slicing it into thin slices perpendicular to the height, c, axis and using the method discussed in Section 11−3. Obtain a formula valid for an arbitrary number, N, of slices and then find the limit of this formula as $N \to \infty$.

   b. Slice the pyramid as above, but find the volume by evaluating an integral analytically as in Section 11−4

   c. Slice the pyramid as above, but find the volume numerically, using a first-order method similar to that discussed in Section 11−2. Design your worksheet so that the values of the dimensions, a, b, and c can be specified by entering them into cells of the worksheet. Experiment with whatever values of these dimensions you like, but turn in the results for $a = 3$, $b = 2$, and $c = 2$.

   d. Add one or more columns to your worksheet to also evaluate the volume using a centered, second-order numerical method as in Section 11−2.

   e. Compare the results of the two numerical methods with the analytical result you obtained above. Experiment with whatever values of N you like, but make a table showing the errors for the two methods at least for $N = 10$, $N = 20$, $N = 40$, and $N = 100$. Check to see if the errors of the two methods vary with N as expected.

2. The water tank in Notrees is not full, and this question involves figuring out how much water there is in the tank.

   a. If the top of the water in the tank is 12 meters above the bottom of the tank, how much water in cubic meters is there in the tank? Use Quattro Pro to answer the question by slicing up the tank as in the notes and adding up the volume of the appropriate slices.

   b. How much water is there in the tank, measured in gallons?

   c. Develop an analytic formula for the volume of water in the tank expressed as a function of the height of the top of the water in the tank as measured from the bottom of the tank, x. Check your answer with the result you got in the first part of this problem.

3. The use of Monte Carlo integration to find the area of plane figures was discussed in the exercises for Chapter 10. The same idea can be used to find the volume of solid objects such as spheres. Program a worksheet to calculate numerically the volume of a sphere of radius 1 using Monte Carlo integration.

4. Program a worksheet to calculate numerically the function $f(x) = x^3$ between two limits to be placed in cells of the worksheet. The worksheet should implement all three methods discussed in Section 11−6, Euler's method, the rectangular rule, and the trapezoidal rule. Compare the results with the analytic result for the integral. Program the worksheet so that you can easily vary the number of sub-intervals, N, and make a table showing the errors associated with each method for several values of N. Experiment with whatever values of N you like, but include at least $N = 10$, $N = 20$, $N = 40$, and $N = 100$. So that everyone is working on the same problem, integrate $f(x)$ between the limits 1 and 2 for your table. Check to see if the error depends on the number of sub-intervals as expected for the three methods.

5. Consider the following integral $\int_0^1 e^{\sqrt{x}}\,dx$.

   a. Develop a formula based on the trapezoidal rule to do the integral numerically.

   b. Choose $\Delta x = 0.25$ and apply this formula "by hand."

   c. Program Quattro Pro to evaluate the formula, and do so for several values of Δx, including 0.25, 0.05, and 0.01.

   d. It happens that the value of the integral is exactly 2. Use this fact to determine the error in the numerical results in the previous part of this problem for all the values of Δx you used. Based on this information, what is the order of the method?

6. Use Quattro Pro to check on the validity of Eq. (11−7) for values of all N between 1 and 50 by programming a worksheet to evaluate the sum directly.

7. To calculate the volume of a sphere one can divide it differently than I did in the notes. Divide a sphere into a set of thin, concentric, spherical shells and use this division to calculate the volume of the sphere both numerically and analytically as discussed in Sections 11−2, 12−3, and 11−4. The volume of a thin spherical shell is approximately the surface area of the shell times its thickness. In case you have forgotten, the surface area of a sphere of radius r is $4\pi r^2$. (For working this problem, just take this formula for the surface area as a given. The same ideas as discussed in the notes can be used to derive it as well.)

8. In the Chapter, the constants a, b, and c in Eq. (11−19) were evaluated by making use of Eq. (11−20). The constants can also be determined by requiring Eq. (11−18) to work for four value of N. Determine values for a, b, c, and d by requiring Eq. (11−18) to hold for $N = 1$, 2, 3, and 4.

   Notice that this technique assumes $S_2(N)$ to be a cubic in N, as in Eq. (11−19), and that was only a guess. Thus if Eq. (11−19) is correct, then the values for a, b, c, and d you just determined are correct (assuming you made no arithmetic mistakes!), but the technique does not show that Eq. (11−19) holds for values of N other than the ones you used. The technique in the Chapter, on the other hand, shows that Eq. (11−19) and, therefore, the values of a, b, c, and d are correct for all values of N.

9. In this problem, you are to estimate the total weight of the air in the Earth's atmosphere. The density of molecules in the atmosphere depends on the type of molecule, and on the height above the Earth. To a fair approximation, the density (number per unit volume) is given by

$$\rho(x) = \rho_0 e^{-mgx/kT}$$

where x is the height above sea level, m the mass of the molecules, g the acceleration of gravity, k is Boltzman's constant, T is the temperature in degrees Kelvin, and ρ_0 is the density of the molecules at $x = 0$ (i.e. at sea level). For x in meters, room temperature, and an atmosphere composed entirely of nitrogen molecules, the value of $\frac{mg}{kT}$ is about 1.1×10^{-4}, and ρ_0 is about 2.7×10^{25} molecules per cubic meter. You will also need to know that the radius of the Earth is about 6.4×10^{6} meters, and the weight of a nitrogen molecule is about 4.68×10^{-26} kg, which is about 1.0×10^{-25} pounds.

a. Assuming the temperature in the atmosphere remains constant at room temperature how many molecules are there in the atmosphere of the Earth? Calculate the answer both numerically, and analytically.

SOME ADVICE: If the density of molecules were the same at all heights, you could answer the question by just multiplying the density by the volume of the atmosphere. The density is not even approximately constant, so that won't work. The density is nearly constant, however, in a thin spherical shell like the ones used in the previous problem, so you could use the ***Main Idea of Integral Calculus*** to solve the problem. For each shell, you could approximate the number of molecules in it by multiplying the volume times the density at the height of the shell, and then you could add up the number of molecules in all the shells. At first glance, that would appear to be a formidable task because the atmosphere has no top (it would seem that even for a finite shell thickness you would have to consider an infinite number of shells). Fortunately, however, the density falls off very rapidly with altitude, so the shells far out contribute negligibly, and you can stop the sum with a finite number of shells. For the analytic calculation, you may need the following integral

$$\int x^2 e^{-ax}\, dx = -\frac{e^{-ax}}{a}\left[x^2 + 2\,\frac{x}{a} + \frac{2}{a^2}\right]$$

b. How much does the atmosphere weigh? (This part's easy!)

c. The air pressure at sea level is about 14.7 lb/in^2. From your answer in the last part you can estimate what the pressure should be. Comparing the two gives a check on whether or not the whole thing makes sense. How well do they agree?

HINT: Pressure is force (weight) per unit area.

d. One of the reasons this estimate is only an approximation is that the temperature of the atmosphere is not a constant which altitude. It varies considerably. If you knew how the temperature varies with altitude, explain how you would use that information to improve the accuracy of the calculation.

12. FINITE DIFFERENCES AND DIFFERENTIATION

As with the last chapter, I have two main goals in this chapter. First, I want to solidify your understanding of the differential calculus, and second I want to show you some ideas about using a computer to solve problems involving derivatives. Also as with the last chapter, we will use Quattro Pro to provide a numerical solution to problems for which reasonable (sometimes simple) analytic solutions exist. If our only purpose were to solve the problems, it would be dumb not to use the analytic solution. Our goal, however, is to look at the main idea behind the differential calculus from a somewhat different point of view than you probably saw it in a Math course. Choosing problems for which exact, analytic solutions exist helps to make connection between the numerical and analytic methods and to see how well the numerical solution works. I also want to show you some of the ideas behind the numerical methods. These ideas are not often used to evaluate derivatives numerically for reasons I'll discuss at the end of the chapter. The ideas are important, though, because they form the basis for most methods for solving differential equations numerically, and that is a frequent application of computers in engineering. The numerical solution of differential equations is the topic of the next chapter.

12.1 Velocities from a Table of Distances

Molly Malone is working at the gun test facility of the Naval Surface Warfare Center. In one test a big gun is fired almost straight up and the height of the projectile is measured every half second. The whole thing is computerized, and the result of a shot is a pretty long table of times and heights. Table 12−1 shows the first 11 entries in one such table of data Molly acquired.

Time (Seconds)	Height (Meters)
0.0	0.0
0.5	497.5
1.0	990.1
1.5	1477.8
2.0	1960.7
2.5	2438.6
3.0	2911.8
3.5	3380.1
4.0	3843.7
4.5	4302.4
5.0	4756.6

TABLE 12−1. Molly's Data for Projectile Height vs. Time.

One of the things Molly must do to analyze the results of such a shot is to determine the speed of the projectile at each time. If something is moving with a constant speed, then that speed is the distance it moves divided by the time it took to do so. Unfortunately, the projectile almost certainly is not moving with constant speed because we expect it to slow down as it rises, finally stopping briefly at the highest point and then turning around to fall back toward the Earth. (Well actually the river—they conduct the tests in the Potomac River.)

If the speed isn't changing very much during each half second interval, we won't make very much of an error if we approximate the speed during that interval by the distance traveled divided by the time, half a second in this case. Table 12−2 shows the result of doing that. The velocity at the start of the i^{th} time

interval, v_i, is approximated as

$$v_i \approx \frac{y_{i+1} - y_i}{\Delta t} \tag{12−1}$$

where the y's are the height readings, and Δt is the time between measurements, 0.5 second in this case.

Time (Seconds)	Height (Meters)	Approx. Speed (m/sec)
0.0	0.0	995
0.5	497.5	985.2
1.0	990.1	975.4
1.5	1477.8	965.8
2.0	1960.7	955.8
2.5	2438.6	946.4
3.0	2911.8	936.6
3.5	3380.1	927.2
4.0	3843.7	917.4
4.5	4302.4	908.4
5.0	4756.6	

TABLE 12−2. Projectile velocity from Molly's data in Table 12−1.

Looking at Table 12−2, we see that the speed is decreasing with each measurement, so the speed was actually a little bigger at the start of each time interval than it was at the end. The data in the third column of Table 12−2 are actually a better approximation to the average speed during the interval, than to the speed at the start of the interval as implied in the table. The first entry, for example, is more accurately the speed at 0.25 s than at 0.0 s. Similarly, the second entry is more accurately the speed at 0.75 s than 0.5 s as implied in the table. If we really want the speed at 0.5 s, perhaps we would be better off to take the average of the first and second entries—that is of the speeds at 0.25 and 0.75 s. Thus in this second approximation we would say that the speed at 0.5 s is more nearly $\frac{1}{2}(995 + 985.2) = 990.1$.

The derivation of a general formula for this improved approximation is interesting. According to the ideas in the preceding paragraph, the speed halfway through the interval is approximately

$$v_{i+\frac{1}{2}} \approx \frac{y_{i+1} - y_i}{\Delta t} \tag{12−2}$$

and the speed at the start of a given interval is the average of the speeds at the half-intervals on either side,

$$\begin{aligned} v_i &\approx \tfrac{1}{2}(v_{i+\frac{1}{2}} + v_{i-\frac{1}{2}}) \\ &\approx \frac{1}{2}\left(\frac{y_{i+1} - y_i}{\Delta t} + \frac{y_i - y_{i-1}}{\Delta t}\right) \\ &= \frac{y_{i+1} - y_{i-1}}{2\Delta t} \end{aligned} \tag{12−3}$$

The result of applying Eq. (12−3) to Molly's data is compared to that of Eq. (12−1) in Table 12−3.

Eq. (12−3) says that the speed is approximately the distance traveled in two time intervals divided by $2\Delta t$, with the two intervals flanking the time at which we want the speed on either side. It seems surprising that we would get a better approximation to the speed by taking twice the time interval than by using a

Time (Seconds)	Height (Meters)	Eq. (12−1) Approx. Speed (m/sec)	Eq. (12−3) Approx. Speed (m/sec)
0.0	0.0	995	
0.5	497.5	985.2	990.1
1.0	990.1	975.4	980.7
1.5	1477.8	965.8	970.6
2.0	1960.7	955.8	960.8
2.5	2438.6	946.4	951.1
3.0	2911.8	936.6	941.5
3.5	3380.1	927.2	931.9
4.0	3843.7	917.4	922.3
4.5	4302.4	908.4	912.9
5.0	4756.6		

TABLE 12−3. Two approximations for the projectile velocity from Molly's data in Table 12−1

single interval. The reason is the centering or lack of it. The speed in Eq. (12−1) is most accurately the speed halfway through the interval, but the equation claims it is the speed at the start of the interval. That's not quite right. The right-hand side of Eq. (12−3), however, is approximately the speed at the time halfway through the interval $t_{i-1} \rightarrow t_{i+1}$, which is the time t_i, as the equation claims. It is usually true that it is better to correct the centering problem at the expense of increased error due to taking a larger time interval, than to live with the non-centering error.

In analogy with the discussion in the previous chapter, Eq. (12−1) is a first order method, and Eq. (12−3) is second order, for similar reasons. The first order method is lop-sided, the second is centered. I'll not demonstrate it to you with this example, but the error with both methods decreases with decreasing Δt. The difference is that with the first order method, the error is proportional to Δt, whereas with the second it is proportional to $(\Delta t)^2$. In a later section I'll be able to show that to you. Eq. (12−1) is said to be *forward-time*, and Eqs. (12−2) and (12−3) *centered-time*. Notice that the only difference between Eqs. (12−1) and (12−2) is that one claims to be the velocity at the start of the time interval, and the other the velocity at the middle. The right-hand sides of the two equations are the same.

Anyway, I'd now like to put this into a worksheet so that I can change things without having to do a lot of arithmetic by hand, and so that I can make use of the graphics capability of Quattro Pro. You will probably not be surprised by my admission that I made up the stuff about Molly and the gun tests. (If not, I have some interesting property on a bridge just outside Brooklyn that you might be interested in buying.) If I were to continue the fiction, I'd have to provide a long table with a few hundred entries giving the measured heights vs. time, and then to get it into Quattro Pro you would have to painfully copy it into a worksheet. Instead, I'll just come clean and admit that I made it up, and the data in Table 12−1 came from an equation which should describe the motion of the projectile assuming that the effect of air friction can be treated fairly simply. The equation is

$$y = \frac{m}{\gamma}\left(v_0 + g\,\frac{m}{\gamma}\right)\left(1 - e^{-\frac{\gamma}{m}t}\right) - g\,\frac{m}{\gamma}\,t \qquad (12-4)$$

where m is the mass of the projectile (in kg), γ is a number describing the air resistance (the larger γ, the larger the air resistance), v_0 is the speed of the projectile when it leaves the end of the gun (assumed to be $t = 0$), and g is a constant related to the strength of gravity, and is 9.8 m/sec^2. For the purposes of this discussion, you should take Eq. (12−4) as just an empirical fact, and not worry about where it came from. If

you are interested, and have the math and physics background, Eq. (12−4) is the solution of the "$F = m\,a$" equation,

$$m\,\frac{d^2y}{dt^2} = -\,mg - \gamma\,\frac{dy}{dt} \tag{12−5}$$

subject to the initial conditions $y(0) = 0$, and $y'(0) = v_0$.

To generate the heights in Table 12−1, I used Eq. (12−4) with $\Delta t = 0.5$ sec, $m = 1$ kg, $\gamma = 0.01$ kg/sec, $v_0 = 1000$ m/sec, and $g = 9.8$ m/sec^2. Instead of giving a table of 100 or so values of y_i to painfully copy into Quattro Pro, I'll cheat a little and just use this equation with these parameter values to generate the table of height vs. time in the worksheet.

Here's how I programmed the worksheet. I put the times in column `A`, the heights in `B`, the forward-time approximation to the speed in `C`, and the second order, centered-time approximation in `D`. In `A7` I put the value of Δt, 0.5 in this case, the mass m in `B7`, γ in `C7`, v_0 in `D7`, and g in `E7`. To make Eq. (12−4) a little less messy to enter into the worksheet, I put $\frac{m}{\gamma}$ in `F6`, and $\frac{m}{\gamma}(v_0 + g\,\frac{m}{\gamma})$ in `F7`. To generate the time column of the table, I put `0` in cell `A11`, `+A11+$A$7` in `A12`, and copied `A12` into the range `A12..A360`. For the height data, I put the formula `$F$7*(1-@EXP(-A11/$F$6))-$E$7*$F$6*A11` in `B11`, and copied this formula into the range `B12..B360`.

Before continuing, use the graphing capability of Quattro Pro to graph the height vs. time. You should find that the result is a slightly lop-sided, upside-down parabola. Notice that Eq. (12−4) says that the projectile continues going down when it strikes the Potomac river (when $y = 0$). That's because no one told it about the river being there. To include this feature, you would have to increase γ substantially when the height becomes 0. (If you have ever done a "belly flop" into a swimming pool, you should appreciate difference between the resistance of water and of air to motion.) We could do that by adding a `@IF` statement into the formulae in column `B` to test whether or not y is less than 0. I suggest you avoid the complication by just limiting the range of the data to be plotted to those entries for which $y \geq 0$, or else just ignoring points with $y < 0$. Once you have everything working, play around with the values of γ and see if you can make sense of the results.

Back to the problem of finding the speed of the projectile, in column `C` we'll put the forward-time, first order approximation for the speed, and in column `D` the centered-time, second order approximation. In `C11` I put the formula `(+B12-B11)/$A$7`, copy it down the column. A disadvantage of the centered-time, second order formula in Eq. (12−3), is that it can't be used for the first item in the table. (Both methods can't be used for the last item.) If the speed at the first time is required, one can either use the first order, forward-time approximation, or extrapolate the table of known speeds to $t = 0$. These are significant complications, and I don't want to get into them here. Fortunately, in this case, I can cheat because I know that the parameter, v_0, stored in cell `D3`, is the speed at $t = 0$. Therefore, in cell `D11` I just put `$D$7`, in `D12` I put the centered-time formula, `(+B13-B11)/(2*$A$7)`, and I copied this formula down the table. Finally, I put in some labels and some lines. Fig. 12−1 shows the worksheet I came up with.

Looking at the speeds in either column, you should find that the results are reasonable, with the centered-time method giving speeds about 5 m/sec higher at small times than does the forward-time method. That's expected since the forward-time method gives more closely the speed halfway through the interval than at the start of it. Also note that the forward-time method gives a speed of about 995 m/sec at $t = 0$ sec, rather than the correct value of 1000 m/sec. Again that's the expected behavior since the forward-time result is closer to the speed at $t = 0.25$ sec than at 0. I suggest graphing the speed along with the height. To do that, you'll probably want to use a different vertical scale for the speeds than for the heights. You can do that by selecting the `Customize Series` item from the `Graph` menu, then selecting the `Y-Axis` from the sub-menu, and then specifying the `Secondary Y-Axis` for the `2nd Series`. The default settings for this axis should be OK, but if you want to change them, you can access them by choosing the `Y-Axis`

a)

	A	B	C	D	E	F
1	Chapter 12: Numerical calculation of velocity using					
2	both forward and centered differences					
3						
4						
5			PARAMETERS			CONSTANTS
6	dt	m	gamma	v0	g	+B7/C7
7	0.5	1	0.01	1000	9.8	+F6*(D7+E7*F6)
8						
9				Velocity		
10	t	x(t)		Fwd.	Ctd.	
11	0	+$F$7*(1-@EXP(-A11/$F$6))-$E$7*$F$6*A11		(+B12-B11)/$A$7	+D7	
12	+A11+$A$7	+$F$7*(1-@EXP(-A12/$F$6))-$E$7*$F$6*A12		(+B13-B12)/$A$7	(+B13-B11)/(2*$A$7)	

b)

	A	B	C	D	E	F
1	Chapter 12: Numerical calculation of velocity using					
2	both forward and centered differences					
3						
4						
5			PARAMETERS			CONSTANTS
6	dt	m	gamma	v0	g	100
7	0.5	1	0.01	1000	9.8	198000
8						
9				Velocity		
10	t	x(t)		Fwd.	Ctd.	
11	0	0		995.0582	1000	
12	0.5	497.5291		985.2076	990.1329	
13	1	990.1329		975.4061	980.3068	
14	1.5	1477.836		965.6535	970.5298	
15	2	1960.663		955.9495	960.8015	
16	2.5	2438.637		946.2939	951.1217	

Figure 12−1. The first few rows of the worksheet I programmed to calculate the forward and centered difference approximations to the bullet velocity. Part a) shows the formulae in each cell, and part b) shows the worksheet as it appears on the screen.

item from the `Graph` menu, and then the `2nd Y-Axis` item from that sub-menu.

I suggest you experiment with different values of γ and perhaps v_0. Notice that the speed falls to zero at the top of the path of the projectile, as expected, and that the speed when it strikes the Potomac on the way back down is less than it was when it left the gun due to air resistance. If you make γ very small, you should find that the speed at the end becomes more nearly equal to the speed at the start.

12.2 Connection with Derivatives

In the previous section, we were interested in finding the *rate of change* of the projectile height with time. Finding the rate of change of one quantity with respect to another is a common problem. For example, in electrical circuits the voltage across a coil of wire (called an inductor) is proportional to the rate of change of the current through the coil with time, or the flow of heat from one point in a long thin rod is proportional to the rate of change of temperature of the rod with position along the length of the rod. The problem of finding the rate of change of one quantity with respect to another is so common that it is given a name. The rate of change is said to be the *derivative* of the first quantity with respect to the second.

For ease of discussion, let's call $f(x)$ the quantity of interest, and assume that we want the rate of change of f with respect to x, or in "mathematicalese" the derivative of f with respect to x. There are several mathematical notations for the derivative of f. Probably the most common is due to Leibniz, and is $\frac{df}{dx}$. Other common notations are $f'(x)$, and $\dot{f}(x)$. If the only knowledge we have of $f(x)$ is some table of values of f for a discrete set of values of x, there is little more we can do than some procedure such as outlined in the previous section. If, on the other hand, we know the functional form of $f(x)$ (for example maybe $f(x) = x^4$), then there is quite a bit more we can do. If we want an approximation to the derivative of f at some value of x then as in the previous section we can write (remembering that the derivative is just the rate of change)

$$\frac{df}{dx} \approx \frac{f(x+\Delta x) - f(x)}{\Delta x} \tag{12−6}$$

where Δx is some small increment of x we choose arbitrarily. This is just the "forward-x" approximation of the previous section. The equation is only an approximation because the rate of change of f is not constant throughout the interval $x \to x+\Delta x$. The approximation can be made more accurate by choosing Δx smaller since the rate of change of f will be more nearly constant over a smaller interval.

Here's the bright idea Newton first thought up. Eq. (12−6) is only approximately correct, but it can be made more and more accurate by choosing smaller and smaller values of Δx. If we could take $\Delta x = 0$, it would become exact. We can't do that because the equation becomes $\frac{0}{0}$, but we can calculate the approximate values of $\frac{df}{dx}$ for several, decreasing values of Δx, and extrapolate to the value the result approaches as $\Delta x \to 0$. That's the bright idea that Newton had. He defined the derivative as being exactly

$$\frac{df}{dx} = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x} \tag{12−7}$$

So Sir Isaac had a bright idea, but how do you implement it? Sometimes it's easy, sometimes difficult, and sometimes very difficult. I'll look at one example which is easy. Consider the example $f(x) = x^4$. Then

$$\begin{aligned}
\frac{df}{dx} &= \lim_{\Delta x \to 0} \frac{(x+\Delta x)^4 - x^4}{\Delta x} \\
&= \lim_{\Delta x \to 0} \frac{(x^4 + 4x^3\Delta x + 6x^2\Delta x^2 + 4x\Delta x^3 + \Delta x^4) - x^4}{\Delta x} \\
&= \lim_{\Delta x \to 0} \frac{4x^3\Delta x + 6x^2\Delta x^2 + 4x\Delta x^3 + \Delta x^4}{\Delta x} \\
&= \lim_{\Delta x \to 0} 4x^3 + 6x^2\Delta x + 4x\Delta x^2 + \Delta x^3 \\
&= 4x^3
\end{aligned} \tag{12−8}$$

This is the idea behind determining the derivative of any function. To calculate the derivative analytically, one needs to find the limit as Δx approaches 0 of the right hand side of Eq. (12−7). For functions which are a power of x this is pretty easy, for other functions such as sines and cosines it's somewhat more difficult, and for others such as Bessel or hypergeometric functions, it's a lot more difficult. Fortunately, the differential calculus has been known for a long time, and people have worked out the derivatives of most functions and put them in tables. These along with some of the theorems you learned in Math 106 are sufficient to evaluate the derivatives of any function you are likely to come across.

You may be wondering why in the definition of the derivative, Eq. (12−7), the forward-x approximation is used rather than the centered-x approximation which seemed to work so much better in the stuff we did in the previous section. The answer is that either approximation works since we are taking the limit as $\Delta x \to 0$, in which case both become exact. The forward-x version is simpler algebraically, so that is the most commonly used. When evaluating derivatives numerically, on the other hand, we can never take $\Delta x = 0$, so the centered-x approximation usually gives the better approximation.

12.3 The Numerical Evaluation of Derivatives

Partly because analytic formulae for the derivatives of most functions are known, it is unusual to evaluate the derivative of a function numerically. I will use Quattro Pro to calculate the derivatives of a simple function numerically at several points, just to illustrate some of the ideas in Section 12−1. As with the integrals in the previous chapter, if my only interest were in the value of the derivative of the function, it would be dumb to do it this way.

For illustrative purposes, I'll use both forward-x and centered-x methods to approximate numerically the derivative of the function $f(x) = x^4$ for a value of x which you specify by entering the value in a cell, and for several values of Δx. The forward and centered approximations to $\dfrac{df}{dx}$ are

$$\left(\left. \frac{df}{dx} \right|_x \right)_{\text{fwd–approx}} = \frac{f(x+\Delta x) - f(x)}{\Delta x} \tag{12−9a}$$

and

$$\left(\left. \frac{df}{dx} \right|_x \right)_{\text{ctd–approx}} = \frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x} \tag{12−9b}$$

Make sure you understand why Eq. (12−9a) is a forward-x approximation, and Eq. (12−9b) is a centered-x approximation. Both claim to be the derivative evaluated at the same point, x, but the first uses the values of f at $x + \Delta x$ and at x to make the approximation, whereas the second uses values of f equally spaced on either side of x. If Eq. (12−9a) claimed to be the value of the derivative evaluated at $x + \frac{1}{2}\Delta x$, instead of at x, then it would be a centered-x approximation as well.

Let's program this into Quattro Pro. You might want to make your own worksheet to find the derivative of a different function for which you know the analytic result. I'm particularly interested in showing the relative accuracies of the two methods, and how these accuracies depend on Δx, so I'll set up the worksheet to approximate the derivative using both methods for a set value of x, which can be changed by changing the contents of one cell, and for a range of values of Δx. I'll put the values of Δx in column A, the forward-x approximation to the derivative in column D, the centered-x approximation in E, the forward-x error in F, and the centered-x error in G. I chose to start these quantities in row 11, leaving room for a title and for the parameters just above the table. I put the value of x for which the derivative is to be approximated in A7 (I started with x=1). I'd like to start the values of Δx at 0, but I can't do that because it would generate a division by zero error. Instead of starting at 0, I'll start at some small number, which I'll specify in cell B7 (I chose 0.0001), and I'll calculate derivatives for a set of values of Δx spaced by an

increment specified in C7 (I chose 0.001). Once the values of these parameters are specified, the exact values of $f(x)$ and of the derivative, $\frac{df}{dx}$ are determined. I will need these values in the worksheet, so I put them in cells E7 and F7.

To calculate the two approximations to the derivatives In A11 I put $B$7, in A12 I put $C$7, and in A13 I put A12+$C$7. I'll need for each value of Δx, besides the value of $f(x)$, the values of $f(x-\Delta x)$ and $f(x+\Delta x)$. I chose to put these values in separate columns, B and C respectively. In B11 I put ($A$7-A11)^4, and in C11 I put ($A$7+A11)^4. In cell D11 I put the formula for the forward-x approximation, (+C11-$E$7)/A11, and in E11 the formula for the centered-x approximation, (+C11-B11)/(2*A11). In cells F11 and G11 I put the errors with the two approximations. In cell F11 I put $F$7-D11, and in G11 I put $F$7-E11. To fill out the table, I copied the range B11..G11 to B12..G12, and I copied the range A12..G12 down as far as I wanted to go. I chose to use 20 values, so I copied to the range A13..G30. Fig. 12−2 shows the worksheet.

It is instructive to graph the results. Use the graphing capability of Quattro Pro to graph the errors for the two approximations vs. Δx. You could put them on separate graphs, but I suggest putting them on the same one. If you do that, however, you will have to use a different y-axis scale for the two quantities. You can specify that by choosing the Y-Axis item from the Customize Series sub-menu. Notice that, as promised way back in Section 12−1, the forward-x method is first order since the error is a linear function of Δx (it graphs to a straight line). The centered-x method seems to be a second-order method since the error graphs to what looks like a parabola, but it's hard to be sure. If you look at the numbers on the worksheet, you will see that the error scales exactly as the square of Δx. Therefore, the centered-x method is second order.

Before leaving this subject, I'd like to show you why taking derivatives of experimental data numerically is a tricky undertaking. The difficulty is that numerical approximations such as Eq. (12−1) or Eq. (12−3) work pretty much as advertised as long as the data in the table, the y_i's, are exact or at least very accurate. In Eq. (12−3), for example, we will want to take Δt small (corresponding to a short time interval between readings) in order that it be a good approximation to the derivative. But in this case the two values of y, y_{i+1} and y_{i-1} will differ only slightly. The difference in the numerator of Eq. (12−3) will then be the small difference of large numbers, and small errors in either of the data values will result in a much larger error in the estimated derivative.

You can use Quattro Pro to demonstrate this effect. The idea is to take some simple function, such as $f(x) = x^4$, add a little noise to it using @RAND, and then calculate, say, the centered-time approximation to the derivative using the noisy values. Here's how I did it, you may want to work it out on your own. In column A I put the x values for my table, in column B I put the exact value of $f(x)$ (I used x^4), and in column C I put the noisy values. In columns D and E I put the derivatives calculated with the exact and noisy values of f, respectively. I chose to start my table in row 11, and to put column labels in row 10. In cell B7, I put the value of Δx to be used, in cell C7 I put the starting value of x, and in D7 the magnitude of the noise. To start, I used $\Delta x = .01$, the starting value of x was 1, and I chose a noise magnitude of 0.1.

The specific formulae I used are as follows. In A11 I put $C$7 and in A12 I put +A11+$B$7. I then copied this formula down as far as I wanted to go. I chose to go to $x = 3$, so for the chosen value of Δx, that meant copying it down to A210. For columns B and C, I put +A11^4 in B11 and +B11+$D$7*(@RAND-0.5) in C11, and then I copied these two cells down the worksheet. For the derivatives, I chose to use the centered x approximation, so in D12 I put (+B13-B11)/(2*$B$7) and in E12 I put (+C13-C11)/(2*$B$7), and copied these cells down the worksheet. After doing all this, you should find that there is much more noise in the "noisy" derivative values than in the "noisy" function values. The point is made especially clearly by making a graph. I suggest you either graph both function values on the same graph and both derivative values on another, or graph all four on the same graph, using the secondary y-axis for the derivative values. Fig. 12−3 shows the "clean" and "noisy" versions of the function, and the derivatives. The "noisy" version of the function is offset upwards by 2 to separate the two

a)

	A	B	C	D	E	F	G
1	Chapter 12: Dependence of errors in forward and						
2	centered difference approximations on dx.						
3							
4							
5	PARAMETERS				CONSTANTS		
6	x	dx min	delta dx		f(x)	f'(x)	
7	-1	0.0001	0.001		+$A$7^4	4*A7^3	
8							
9				f'(x)		Error	
10	dx	f(x-dx)	f(x+dx)	Fwd.	Ctd.	Fwd.	Ctd.
11	+$B$7	(+$A$7-A11)^4	(+$A$7+A11)^4	(+C11-$E$7)/A11	(+C11-B11)/(2*A11)	+$F$7-D11	+$F$7-E11
12	+$C$7	(+$A$7-A12)^4	(+$A$7+A12)^4	(+C12-$E$7)/A12	(+C12-B12)/(2*A12)	+$F$7-D12	+$F$7-E12
13	+A12+$C$7	(+$A$7-A13)^4	(+$A$7+A13)^4	(+C13-$E$7)/A13	(+C13-B13)/(2*A13)	+$F$7-D13	+$F$7-E13

b)

	A	B	C	D	E	F	G
1	Chapter 12: Dependence of errors in forward and						
2	centered difference approximations on dx.						
3							
4							
5	PARAMETERS				CONSTANTS		
6	x	dx min	delta dx		f(x)	f'(x)	
7	-1	0.0001	0.001		1	-4	
8							
9				f'(x)		Error	
10	dx	f(x-dx)	f(x+dx)	Fwd.	Ctd.	Fwd.	Ctd.
11	0.0001	1.0004	0.9996	-3.9994	-4	-0.0006	4E-08
12	0.001	1.004006	0.996006	-3.994	-4	-0.006	4E-06
13	0.002	1.008024	0.992024	-3.98802	-4.00002	-0.01198	1.6E-05
14	0.003	1.012054	0.988054	-3.98204	-4.00004	-0.01796	3.6E-05
15	0.004	1.016096	0.984096	-3.97606	-4.00006	-0.02394	6.4E-05

Figure 12−2. The first few rows of the worksheet I programmed to determine the dependence of the errors associated with the forward and centered difference approximations on the step size, Δx. Part a) shows the formulae in each cell, and part b) shows the worksheet as it appears on the screen.

curves.

Because of this problem, one rarely takes numerical derivatives of empirical data in this way. If it is really necessary to find the derivative of some empirical quantity, another approach is usually taken. Very briefly, the idea is to try to fit the empirical data with some analytic function or set of functions which are smooth, and for which analytic derivatives are available. For example, in a situation such as that in Section 12−1, Molly would probably have tried to find a polynomial of some order which fit the empirical data as well as possible, and then taken the derivative analytically of this polynomial. The idea behind how to find

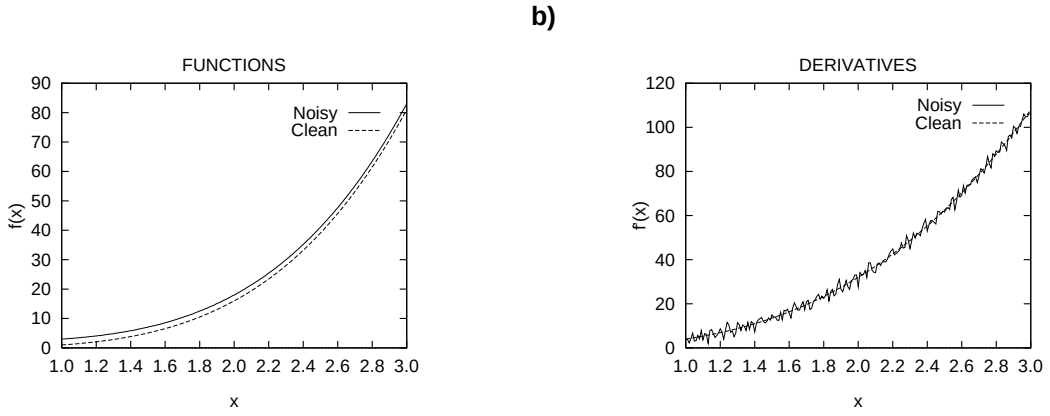

Figure 12−3. Plots showing the effects of a small amount of noise on numerical derivatives. Part a) shows the function x^4 (Clean), and the same function, but with a random noise of amplitude ± 0.05 added (Noisy). The Noisy curve has been offset slightly to separate it. Part b) shows the centered difference approximation to the derivative of the Clean and Noisy functions, using the x increment $\Delta x = 0.01$.

such a polynomial is just the same as that Herbie Husker encountered in trying to analyze his current vs. voltage data back in Chapter 6. The main difference is that Herbie chose a first order polynomial (a straight line), and in this case Molly would certainly want to choose a polynomial of higher order, like a cubic or quartic.

We are not going to pursue the matter further in these notes, but finding a polynomial of some order which fits a table of data as well as possible is a fairly common problem, and a large volume of mathematics has grown up around it. It turns out that certain sets of rather unlikely appearing polynomials are the most convenient to work with. These sets have been given names, and some are rather famous. There are Legendre polynomials, Laguerre polynomials, Hermite polynomials, and so forth. Probably the most useful polynomials have the hard-to-spell name Chebyshev polynomials. I've seen almost as many spellings of this name as I have books on the subject. Sometimes it starts with a 'T'.

12.4 Exercises

1. Using both the forward-x and centered methods, estimate the derivative numerically of the following function, f:

$$f(x) = 5\sin(3x)$$

for $x = 0.1$ and $x = 1.0$ radians. Using the same general procedure as in Section 12−3, investigate how the accuracy of both methods depends on the value of Δx used in making the numerical estimate. Does the error vary with Δx as expected for both methods?

2. Use the method discussed in Section 12−2 to determine analytically directly from the definition, Eq. (12−7), the derivative of the following function,

$$f(x) = \sin x$$

You may find the following trigonometric identities helpful.

$$\sin(a + b) = \sin a \cos b + \cos a \sin b$$

$$\cos(a + b) = \cos a \cos b - \sin a \sin b$$

The following approximations may also be useful. They become more and more accurate as ε becomes smaller and smaller. (It would be a good idea to use Quattro Pro to check them out.)

$$\sin\varepsilon \approx \varepsilon$$

$$\cos\varepsilon \approx 1 - \varepsilon^2$$

3. Which of the following is the correct result for $\frac{d}{dx}(x \log x^2)$?

   a. $2 + \log x^2$

   b. $2x + \log x^2$

   c. $2(1 + \log x)$

Answer the question first analytically by using the chain rule to evaluate the derivatives, and then check your answer by approximating the derivative for a few specific values of x numerically.

4. In this problem you are to find an approximate formula for the second derivative of a function, $\frac{d^2 f}{dx^2}$, similar to Eq. (12−9b). Use the following idea. Divide up the x axis into an equally spaced grid of points with spacing Δx. Let $g = \frac{df}{dx}$. Then the required second derivative is just the first derivative of g, $\frac{d^2 f}{dx^2} = \frac{dg}{dx}$. If I knew g at points half way in between the grid points, then a centered approximation to $\frac{dg}{dx}$ (and, therefore $\frac{d^2 f}{dx^2}$) at say the i^{th} grid point would be

$$\left.\frac{d^2 f}{dx^2}\right|_i = \left.\frac{dg}{dx}\right|_i \approx \frac{g_{i+\frac{1}{2}} - g_{i-\frac{1}{2}}}{\Delta x}$$

But a centered approximation to $g(x) = \frac{df}{dx}$ at the half-way points is

$$g_{i+\frac{1}{2}} = \left.\frac{df}{dx}\right|_{i+\frac{1}{2}} \approx \frac{f_{i+1} - f_i}{\Delta x}$$

Putting this approximation for the first derivative of g into the approximation for the second derivative of f just above yields the desired formula.

Use Quattro Pro to implement your formula, and test it using the function $f(x) = x^4$ as in Section 12−3 where I tested the forward and centered approximations to the first derivative. By examining the way the error of the method depends on Δx, determine the order of the method.

5. Here's another way to derive the centered approximation to the derivative, Eq. (12−9b). Suppose we want to know the value of the derivative of some function, f, at the i^{th} grid point. Then we find a quadratic polynomial which has the same values as f at the three grid points, x_{i-1}, x_i, and x_{i+1}. In other words, we find the quadratic polynomial, $p(x)$ for which $p(x_{i-1}) = f_{i-1}$, $p(x_i) = f_i$, and $p(x_{i+1}) = f_{i+1}$. We can do so, because there are three unknown constants in a quadratic, and these three requirements result in three equations in these three unknowns. This is the same procedure we used in Section 10-4. Once the quadratic polynomial is known, it can be differentiated analytically, and the derivative evaluated at x_i.

   a. Use this procedure to determine an approximate formula for the first derivative at the i^{th} grid point, and compare your result with Eq. (12−9b).

   b. Once the quadratic is known, it's easy to take the second derivative. Do so, and determine a formula for the second derivative, $\dfrac{d^2 f}{dx^2}$ at x_i.

6. A complicated circuit has a 1 nF (10^{-9} F) capacitor in it as shown in the Fig. 12−1. The voltage across the capacitor, v, is measured at several times, and the results are shown in the table.

t (μsec)	v (Volts)
0	2.00
0.05	1.85
0.10	1.65
0.15	1.41
0.20	1.12

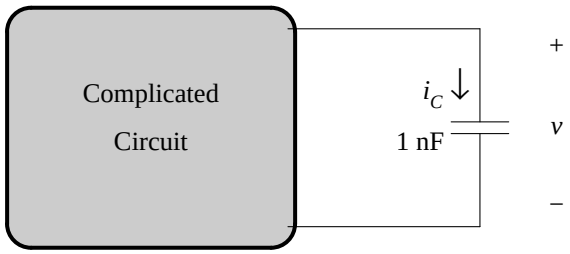

Figure 12−4. Circuit and table of voltages vs. time for Exercise 12−6.

Determine as accurately as you can what the current through the capacitor was at $t = 0.10$ μsec. Explain why you chose the method you used.

13. NUMERICAL SOLUTION OF DIFFERENTIAL EQUATIONS

Many physical laws are expressed most simply as differential equations. A differential equation is an equation which includes derivatives of one or more of the unknowns in the equation. One such example is Newton's Law from physics. If a body is constrained to move along a straight line, m is the mass of the body, F the force acting on it, and x is the position of the body on that line, then Newton's Law states

$$m\,\frac{d^2x}{dt^2} = F(t) \tag{13−1}$$

We briefly encountered two other examples in Chapter 8 in the form of the current-voltage relations for capacitors and inductors. The current into a capacitor, I is related to the voltage across it, V, by

$$C\,\frac{dV}{dt} = I(t) \tag{13−2}$$

and the voltage, V across an inductor is related to the current through it, I, by

$$L\,\frac{dI}{dt} = V(t) \tag{13−3}$$

Recall C and L are numbers which depend on the capacitor and inductor respectively. C is the capacitance, and L the inductance. I could continue to regale you with other examples, but this should give you the idea.

Let's look a little closer at one of these equations, say Eq. (13−2). Dividing both sides by C gives

$$\frac{dV}{dt} = \frac{1}{C}\,I(t) \tag{13−4}$$

where C is a known constant (1×10^{-9} F, for example). Both $V(t)$ and $I(t)$ are functions of time. (I don't explicitly write $V(t)$ in the derivative in Eq. (13−4) because the notation gets messy.) If we know $V(t)$, then finding $I(t)$ is pretty easy: we have only to differentiate $V(t)$ with respect to t and then multiply by C. On the other hand, if $I(t)$ is known, but $V(t)$ is not, then finding $V(t)$ is somewhat more difficult, and the equation is referred to as a *differential equation*. In that case, the problem is to find a functional form for $V(t)$ such that its derivative is equal to a known function, $\frac{1}{C}\,I(t)$. Instead of specifying $V(t)$ directly, this equation specifies the derivative of $V(t)$.

In the discussion of integration and differentiation in the previous two chapters, I could safely assume that you had already seen the analytic part of the material in a math course. For the subject matter of this chapter, that would not be a safe assumption, and I have written the chapter so that it should be accessible even if this is the first time you have even heard of a differential equation. In fact, those who have had some formal introduction to differential equations may be at a disadvantage, because you may be tempted to try using more general methods and concepts to analyze the examples and exercise problems than are necessary. If you are in this group, I recommend that you remember that every differential equation in this chapter can be solved simply. General techniques will work, of course, but for the problems considered here they will often be much more difficult to apply than the simple, but not so general ideas I discuss. If you are pulling out an atomic blunderbuss to attack any of the problems in this chapter, you run the risk of completely demolishing it and perhaps yourself in the mushroom cloud.

I will touch briefly on one method for solving differential equations analytically, but most of the time will be spent on numerical solutions. The analytic solution of differential equations can become quite abstract, and my principal goal in this chapter is to introduce you to differential equations from a concrete viewpoint. It has been my experience that many engineering students have difficulty with differential equations and, I hope that the material in this chapter will help you when you encounter the beasts "for real" in later math or engineering courses.

13.1 Population Growth—A Simple Differential Equation

Consider the population dynamics of a large colony of bacteria. To a pretty good approximation the rate of growth of the colony is proportional to the number of bacteria present. If on the average a bacterium produces another bacterium every T_b seconds, then n bacteria will produce n "child" bacteria during this time interval. If "birth" is the only factor changing the bacteria population, then at any time, t, the population is growing at the rate $n(t)/T_b$ new bacteria per second. The rate of change is the the derivative with respect to time, so a mathematical version of this statement is

$$\frac{dn}{dt} = \frac{n(t)}{T_b} \tag{13−5}$$

Real bacteria also die, but to keep things simple for now lets assume that these are immortal bacteria. I will leave it to you as an exercise to add the grim reaper into the model. Of differential equations that arise in practice, equations of the form of Eq. (13−5) are probably the most common, and they are also among the most friendly. The equation can easily be solved analytically, but I will first try to solve it numerically.

Here's the idea for a numerical solution. What is meant by the term, "birth rate?" If the birth rate is, say 100 per hour, then in one hour 100 bacteria will be "born," and in a time, t, (expressed in hours) $100t$ bacteria will be born. In our problem if the birth rate were a constant, the the number of bacteria at any time would be just the starting number plus the birth rate times the time, with the time expressed in the same units as used for the birth rate. Unfortunately though, our birth rate is not constant. At any instant, t, the birth rate is $n(t)/T_b$, and n is increasing with time.

We encountered a similar quandary in evaluating integrals numerically. The way out of it was to evaluate the integral in a number of small steps. Over each step the value of the integrand was not constant either, but if the step was small enough the integrand didn't change much over the step and approximating it as constant introduced only a small error. The same idea works here as well. If we know the population at some time, say t, we can approximate the population at a slightly later time, $t+\Delta t$, as the population at t plus the birth rate at that time multiplied by Δt. That won't be exactly correct because the birth rate changes during the time step Δt, but by choosing Δt small enough we can make the error as small as we like. Once we have an approximation for $n(t+\Delta t)$, we can use the same idea to obtain an approximate value of the population two time steps later, $n(t+2\Delta t)$, and so forth as far as we like.

That's the idea. Putting it into a formula gives

$$n(t+\Delta t) \approx n(t) + \frac{n(t)}{T_b}\,\Delta t = n(t)\left(1 + \frac{\Delta t}{T_b}\right) \tag{13−6}$$

If we know the population at some time, say t_0, then we can obtain an approximate value for the population at any time by repeated application of this expression. Knowing $n(t_0)$ we use it to obtain a value for $n(t_0+\Delta t)$, then knowing $n(t_0+\Delta t)$ we use it to obtain a value for $n(t_0+2\Delta t)$. Knowing $n(t_0+2\Delta t)$ we use it to obtain a value for $n(t_0+3\Delta t)$, and so forth. The only thing that is needed is knowledge of the population at some time, t_0, so the process can be started off. That's a piece of information the differential equation can't supply. We have to know it from some other source. The differential equation can't know whether we started the colony with just one bacterium or with a whole sneeze-full. This requirement for additional information is a common feature of differential equations, and the required information is called an *initial condition* or a *boundary value*, depending on the problem under consideration.

The result we obtain in this way is only approximate because the equality in Eq. 13−6 is only approximate. This equation can be made more accurate by choosing a smaller time step, Δt, but it is not clear whether the approximate value of n at some specific time obtained in this fashion will be more accurate with smaller time steps. The reason is that although the error per time step decreases as Δt is made smaller, the number of steps required to march a certain distance in time increases, causing the number of times we make the error to increase at the same time. We ran into a similar concern in Chapter 11 with numerical integration. There are ways to show analytically that our method will converge to the correct function in

the limit $\Delta t \to 0$, but I don't want to go into that here. Once we have the method programmed on a spreadsheet we can check "empirically" to see if the results of the calculation approach some set function as Δt is chosen smaller and smaller. In order to let you sleep at night, I'll tell you now that it turns out that the method works.

The method works, but there other methods that work better in that the error decreases faster with step size than does this one. Just as we found for numerical integration schemes, the dependence of the error on the step size is an indication of the *order* of the method. (This use of the term "order" should not be confused with the usage in the term *order of a differential equation* which I will discuss in the next section.) The method we've come up with here turns out to be first order because the overall error is approximately proportional to the first power of the step size. Later we will discuss an improvement on this method for which the error scales as the square of step size.

This solution procedure is a natural for putting on a Quattro Pro spreadsheet, but to do so we have to get more specific and specify values for T_b and $n(t_0)$. Let's consider the case for which $n(0) = 1000$, and $T_b = 50$ min. Since the value of n is specified at time 0, we take $t_0 = 0$, and since T_b is given in minutes, we will measure time in minutes. The value of Δt to be used is somewhat arbitrary—the smaller the value, the more accurate the result will be but the more steps will be required to march to a specified time.

Before I put the whole thing on the computer, I'll march through a few steps manually to clarify what needs to be done in Quattro Pro. For this calculation, I'll choose a fairly large time step of $\Delta t = 10$ min. According to Eq. (13−6) the population after the first time step will be

$$n(10) = n(0) \cdot (1 + \frac{\Delta t}{T_b})$$

$$= 1000 \cdot (1 + \frac{10}{50})$$

$$= 1000 \times 1.2 = 1200$$

Using this value, we can determine $n(20)$,

$$n(20) = n(10) \cdot (1 + \frac{10}{50})$$

$$= 1200 \times 1.2 = 1440$$

and then $n(30)$,

$$n(30) = 1440 \times 1.2 = 1728$$

and so forth, out as far as we want to go.

Doing the arithmetic by hand gets pretty tiresome, and I now want to program Quattro Pro to do the work for me. As usual, I'll tell you how I did the programming, but I suggest you try it on your own before looking at my stuff very closely. I put the values of time in column `A` and the numerical results for the corresponding population in `B`, starting in row `12`. I put the value of $n(0)$ in cell `A7`, the value of T_b in `B7`, and the value of Δt in `C7`. This way, I will be able to change the values of these parameters conveniently.

To generate the table, I put `0` in `A12`, `+A12+$C$7` in `A13`, and `$A$7` in `B12`. I then put a formula corresponding to Eq. (13−6) in `B13`. In doing so, I noticed that the right-hand factor in the equation is the same for each step. Instead of evaluating this factor each time, I put it in cell `E7`. The formula was `1+C7/B7`. Then in `B13` I put `+B12*$E$7`. Finally, I copied `A13..B13` down about 200 rows.

That's it. Fig. 13−1 shows the first few rows of the spreadsheet. Looking at the numbers in column `B`, I see that the population increases explosively. Starting with 1000 bacteria and using a time step of

a)

	A	B	C	D	E	F	G
1	Chapter 13: Numerical calculation of the evolution						
2	of the population of a bacteria colony						
3							
4							
5	PARAMETERS				CONSTANT		
6	n0	Tb	dt		1+dt/Tb		
7	1000	50	1		1+C7/B7		
8							
9							
10		Num.					
11	t	n(t)					
12	0	+$A$7					
13	+A12+$C$7	+B12*$E$7					

b)

	A	B	C	D	E	F	G
1	Chapter 13: Numerical calculation of the evolution						
2	of the population of a bacteria colony						
3							
4							
5	PARAMETERS				CONSTANT		
6	n0	Tb	dt		1+dt/Tb		
7	1000	50	1		1.02		
8							
9							
10		Num.					
11	t	n(t)					
12	0	1000					
13	1	1020					
14	2	1040.4					
15	3	1061.208					
16	4	1082.432					
17	5	1104.081					
18	6	1126.162					
19	7	1148.686					
20	8	1171.659					

Figure 13−1. The first few rows of the worksheet I programmed to calculate the bacteria population. Part a) shows the formulas in each cell, and b) shows the worksheet as it appears on the screen

1 minute, the spreadsheet claims there are more than 50,000 bacteria after 200 minutes. (Fig. 13−1 only shows the first 20 rows, so if you are skeptical or curious, you will have to actually program Quattro Pro to see the row corresponding to t = 200 minutes.) Fig. 13−2 shows a plot of the calculated population vs. time.

This calculation is only an approximation, so how accurate is it? For this differential equation an

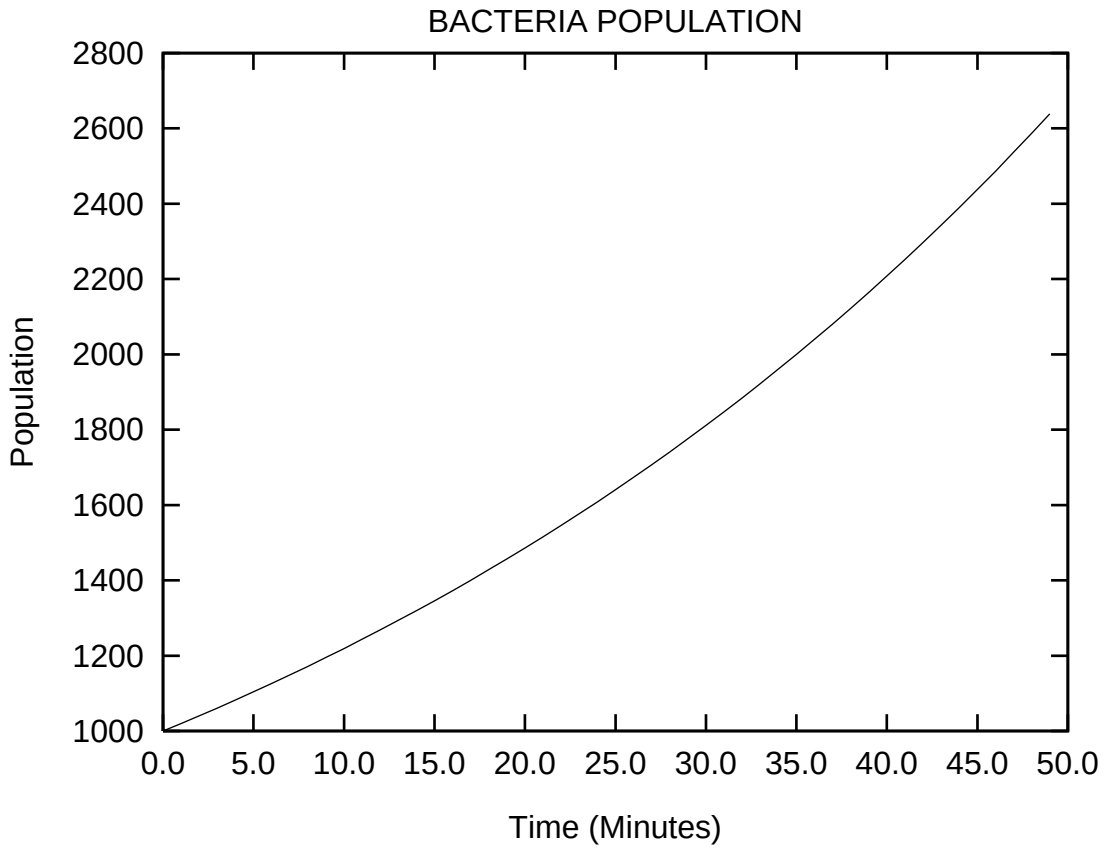

Figure 13−2. Plot of the numerical solution of Eq. 13−5 using the method given in Eq. 13−6. The initial population was $n(0) = 1000$, and $T_b = 50$. The time step was $\Delta t = 1.0$ min.

easy analytic solution exists, and with that we will be able to answer the question directly. I haven't gotten that far yet, so we will have to be satisfied for now with more indirect checks. First, do the results make sense? Looking at the plot, the slope of the curve is constantly increasing. That appears to be correct. The birth rate should be proportional to n, and the larger the birth rate, the more rapidly should n increase. The more rapidly n increases, the more rapidly should the birth rate increase. It's a kind of a "rich get richer" situation.

Our results look reasonable, but do they approach a set function as Δt is decreased? If the method is worth anything, the calculated population function should stabilize as we take Δt smaller and smaller. In making the check, we must be careful to compare values of n at the same time. Remember that a given time occurs at a different step number for different step sizes. Fig. 13−3 shows the result I obtained for time steps of 10, 1, and 0.1 minutes. It appears that our method does converge on a fixed result, as hoped.

(In case you are wondering how I produced Fig. 13−3 on Quattro-Pro, I didn't. Because of the complication of having to plot the values of n at different step numbers for the three curves, I cheated and programmed the same algorithm on a computer using C. I wrote my C program so that it wrote its results to a file, and then I plotted the results. I haven't told you anything about C yet, so I can't tell you any more of the details. You can check up on whether or not I pulled a fast one by comparing values on the curve to values from your spreadsheet for the same time step.)

I'll come back to the topic of the numerical solution of differential equations shortly, but first I want to say something about the analytic solution of differential equations.

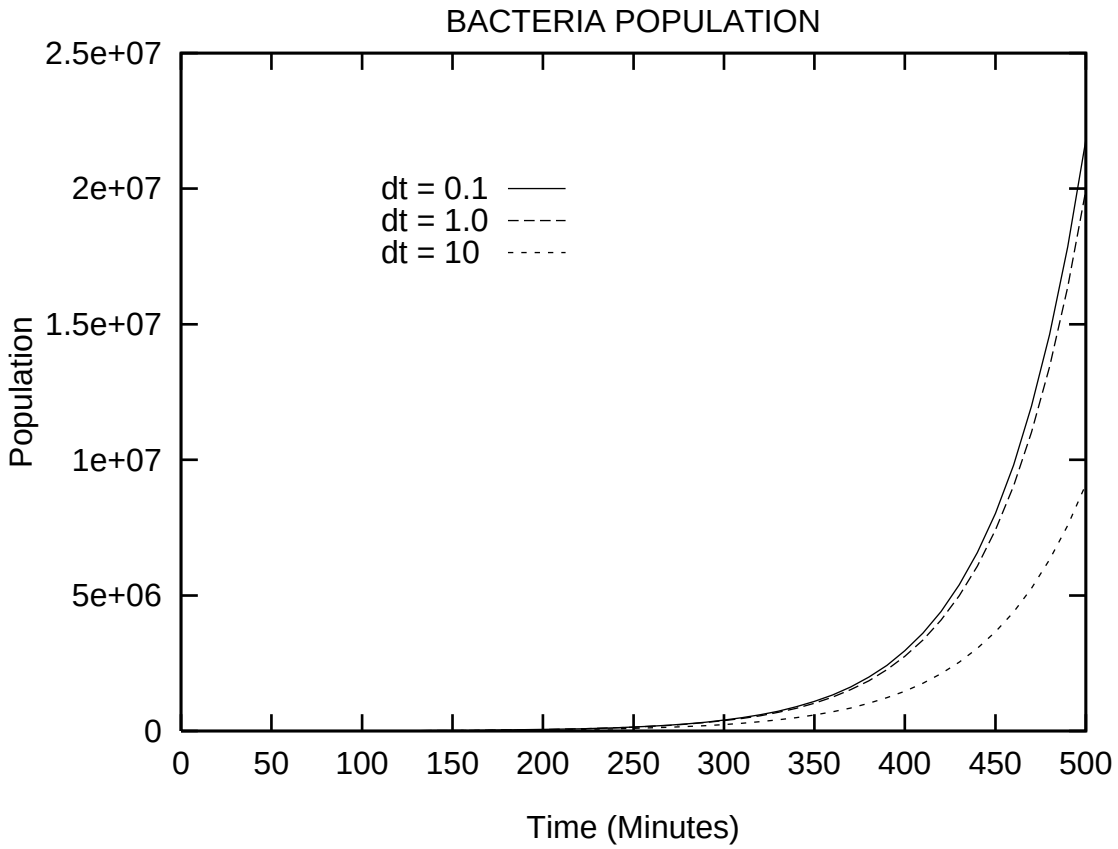

Figure 13−3. Comparison of the performance of Eq. (13−6) in estimating the bacteria population vs. time for three time steps. The parameters used were $n_0 = 0$, and $T_b = 50$ min.

13.2 Analytic Solution of Differential Equations

An analytic solution to a differential equation is almost always preferable to a numerical solution, but many differential equations are very difficult to solve analytically. The topic of how to solve differential equations analytically is one that has occupied mathematicians for more than 100 years and they have had considerable success. Still, for most differential equations the analytic solution is either difficult or unknown. Fortunately, there are a few types of differential equations that are easily solved, and these types appear frequently in practical applications. In this section, I'll first tell you a little about the nomenclature that has developed for describing differential equations, and then I'll show you a simple technique for solving them that only works for a few cases. It turns out, though, that these cases are the ones that appear most frequently in engineering. Even after you have become an expert in the more sophisticated methods of solution, I recommend that you try this one first. If it doesn't work, you haven't lost much; but if it does, you will have saved yourself considerable algebra.

A differential equation is an equation in which one or more terms involve a derivative of one of the unknowns. It differs from an ordinary algebraic equation in that a whole set or table of values of the unknown is desired. For example in the bacteria population problem, the desired solution was the value of the bacteria population for all times, not just the population at a single time. When solving a differential equation analytically, one usually tries to find the functional dependence of the unknown variable on some other variable such as time. For an ordinary algebraic equation, only one or at most a few values are desired. For example, in the discussion of current and voltage in Chapter 8, we wound up with ordinary algebraic equations when analyzing circuits containing voltage sources and resistors. The desired solution

was the current through some branch or the voltage across some element, and only one number was expected. (As we will see shortly, the analysis of circuits can also lead to differential equations. That's one reason for the existence of this chapter!)

Besides this difference in the quantity desired as a solution, there is another difference between ordinary algebraic and differential equations. An ordinary algebraic equation generally has only one, or at most a few, solutions, but a differential equation usually has an infinite number. Here I'm not talking about the infinite number of values necessary to specify *one* solution ($n(t)$ for all possible values of t, for instance), but rather an infinite number of solutions. Typically, the differential equation describes the behavior of a large number of systems, and each of these solutions corresponds to one of these systems. Part of the job of finding a solution then is to decide which of these solutions is the one corresponding to the system you are interested in.

13.2.1 Classification of Differential Equations

Differential equations are classified in several ways, some of which are useful and some of which are not (in my opinion). One useful classification is the *order* of the equation. The order of a differential equation is the highest derivative order that appears in it. Eq. (13−1) is a second order differential equation, and Eqs. (13−2) and (13−3) are both first order. The following viciously unfriendly beast is third order.

$$t^{13}\frac{d^3 f}{dt^3} + f^7\frac{d^2 f}{dt^2} + \log(f) = \sin^{2.5} 3.2t \tag{13−7}$$

The equation is third order because the highest order derivative involving the unknown, f, is third order. (Unfortunately, the same word, "order," is used to mean two different things when applied to the numerical solution of differential equations. One is the highest derivative order in the equation, as just discussed, and the other is the accuracy of the numerical approximation: the "order" of the method. Which meaning is intended is usually obvious from the context, and as long as you are aware of the dual use of the word, it shouldn't be confusing.)

Another useful classification is into *linear* and *non-linear* differential equations. Basically, a linear differential equation is one which can be written in a form in which the unknown appears only as a first power. If the equation isn't linear, it's non-linear. For example, Eq. (13−7) is non-linear because of the seventh power of f multiplying the second derivative in the second term, and because of the $\log(f)$ in the third term. The second term would be non-linear even if it were $f\frac{d^2 f}{dt^2}$ because f times one of its derivatives counts as non-linear. Notice that the linearity of a differential equation does not depend on the functional form of the independent variable, t in the case of Eq. (13−7). The t^{13} in the first term and the $\sin^{2.5} 3.2t$ on the right-hand side of Eq. (13−7) do not make the equation non-linear.

The point of making the linear-nonlinear classification is that linear differential equations are usually pretty well behaved, and the analytic solution of these equations is *relatively* straight-forward. (That doesn't mean it's always easy!) Nonlinear differential equations, on the other hand, are another matter. Some such equations can be solved fairly simply, but usually finding the analytic solution of a nonlinear differential equation is very complicated and difficult. Often, such equations can only be solved numerically, and even then there can be problems. Never turn your back on a non-linear differential equation!

One other useful classification is the grouping into *homogeneous* and *inhomogeneous* equations. A differential equation is homogeneous if all the terms in it involve the unknown, otherwise it's inhomogeneous. Eq. (13−7) is inhomogeneous because of the term on the right-hand side. It would be homogeneous if this term were zero instead. This classification is useful mostly for solving differential equations analytically. A common method for solving an inhomogeneous differential equation involves solving as an intermediate step the related homogeneous equation. We will not discuss such techniques in these notes.

13.2.2 A Method for Solving Some Differential Equations

OK, here it is—my recommended method for solving differential equations. You guess! You are probably thinking something like "Right! If I were sufficiently clairvoyant to do that I'd be spending all my time in Las Vegas!" It really isn't all that bad, and I don't think you will need to buy a pack of Tarot cards or attend any midnight séances. The idea is that you guess a solution, and plug your guess into the differential equation. If it satisfies the differential equation, great. Your worries are over. If it doesn't, you haven't lost much because plugging the guess into the differential equation is usually pretty easy, and it's usually obvious whether or not it works.

Out of all the possible solutions, how can you have any hope of just guessing the right one? I suggest you first guess an exponential, and if that doesn't work try a power of t (or whatever your independent variable is). You will be surprised at how often one of these works. If neither works, then you've probably got trouble.

As an example, I'll use the technique to solve Eq. 13−5 for the time dependence of the bacteria population. Recall, the differential equation is

$$\frac{dn}{dt} = \frac{n}{T_b} \tag{13−5}$$

I will assume that I started the colony off with n_0 bacteria, and for convenience I'll set my clock such that this was done at $t = 0$. I choose to express the initial number of bacteria as a symbol rather than a specific number like 1000. The advantage of doing it this way is that at any stage in doing the algebra it will be easier to see where terms came from. The symbol n_0 will always stand for the initial number of bacteria, but if instead I use a specific number, like 1000, arithmetic may change it to something else, making it lose its identity. At the end I'll substitute 1000 for n_0, so if you find this confusing you might want to remember that n_0 just stands for 1000. For the same reasons, I'll leave the value of the parameter T_b as a symbol for now, even though at the end of all this I'll substitute 50 for it.

Anyway, I'll take my own advice, and guess an exponential for $n(t)$,

$$n(t) \stackrel{?}{=} a\, e^{bt} \tag{13−8}$$

where a and b are constants which can be adjusted so that the guess is a solution of Eq. (13−5). Notice that I guessed a solution with several unknown parameters in it. If there exists a solution of this form, for any values of the constants a and b, the algebra will find it and the values of the constants for me. If I tried to guess the whole thing, including the values of the constants, I almost certainly would fail because there are so many possibilities.

Let's see how well all this works. First, plug the guess, Eq. (13−8) into the differential equation to be solved, Eq. (13−5), giving

$$\frac{dn}{dt} = a\, b\, e^{bt} \stackrel{?}{=} \frac{1}{T_b}\, a\, e^{bt} \tag{13−9}$$

In this equation $a\, e^{bt}$ divides out and we obtain

$$b \stackrel{?}{=} \frac{1}{T_b} \tag{13−10}$$

Remember, b is a constant which is to be chosen, if possible, so that our guess is a solution of the differential equation. According to this result, our guess will work if we can find a value of b such that it is equal to $1/T_b$, and is a constant. Doing that is trivial. Since T_b is itself a constant (50 in this case), the choice $b = 1/T_b$ works just fine. Thus, it appears that

$$n(t) = a\, e^{t/T_b} \tag{13−11}$$

is a solution of my differential equation, for any value at all of the constant a. My method has been more

bountiful than I wanted, however. I set out to find one solution of the differential equation, but instead I came home with an infinite number of them, one solution for each possible value of a.

That's not exactly what I had in mind. Each of the possible solutions is different. I thought I was describing a specific case in which the population of a specific colony was to be determined. At any one time there ought to be one and only one population. It would appear that my math so far is claiming that I can have any population at this time I want by choosing that value of a appropriately. If I choose a to be negative, Eq. 13−11 even predicts a negative population!

What went wrong? The problem is that the differential equation, Eq. (13−5), is very general. It describes *any* group for which the population change is governed solely by birth, with a birth rate of n/T_b. For example, if I start a colony off with 1000 bacteria and you start one off with 106, then if both colonies have the same birth rate, the same equation should describe the population of both. Thus, it is not surprising that Eq. (13−5) has many solutions. There must be a separate solution for each possible colony generated by any possible initial condition.

What all this means is that we are not yet done. The math has provided us with a heap of solutions, one of which is supposed to correspond to my bacteria colony. The problem now is to pull out of this heap that particular solution. How can we do that? Well, what information have we not yet put into the problem that should be needed to specify the solution? The answer is that we have not yet specified the number of bacteria we started the colony with. This information is needed because different starting numbers produce different population functions. We started with n_0 bacteria and the starting time was $t = 0$. (Remember that in this example, n_0 is just 1000.) Thus we require that $n(0) = n_0$. Making this requirement gives us an equation which we hope will pick out the solution corresponding to our problem. Evaluating our general solution at $t = 0$ and making this requirement yields

$$n(0) = a\, e^{0/T_b} = a \cdot 1 = n_0 \tag{13−12}$$

(Remember $e^0 = 1$, not 0.)

Finally then, we have a value for the second unknown constant, a, and the solution is

$$n(t) = n_0\, e^{t/T_b} \tag{13−13}$$

I claim that this is the solution to the differential equation, but you can check by plugging it into the original equation, Eq. (13−5). For the specific case we have have been considering, $T_b = 50$, and $n_0 = 1000$, so the solution is

$$n(t) = 1000\, e^{t/50} \tag{13−14}$$

13.2.3 Comparison of Analytic and Numerical Solutions

Let's compare the answers obtained by the two methods (numerical and analytic). It's easy to modify our spreadsheet to compare the exact analytic result, Eq. (13−14), at each time with the approximate numerical result. Fig. 13−4 shows the results I got for $n_0 = 1000$, $T_b = 50$ min, and for $\Delta t = 10$, 1, 0.1 min. Here I plot the relative difference between the numerical solution for each time step and the analytic. As $\Delta t \to 0$, the difference between the numerical and analytic solution approaches zero. For a fixed time step, we see that changing Δt by a factor of ten results in about a factor of ten change in the error, meaning the the error varies roughly linearly with Δt. This linear dependence of error on time step implies that the method is a first order method. We will see in a later section that this method can be derived from the first order forward difference approximation to the derivative, so this observation makes sense.

Looking at the error results, it appears that my numerical method works. In fact, this numerical method has been known for more than 100 years, and is called *Euler's method*. More accurate methods exist that are only a little more complicated. I will discuss one of these in a later section, and the derivation of another is left as an exercise. For now, though, I want to continue with my discussion of the analytic solution of equations.

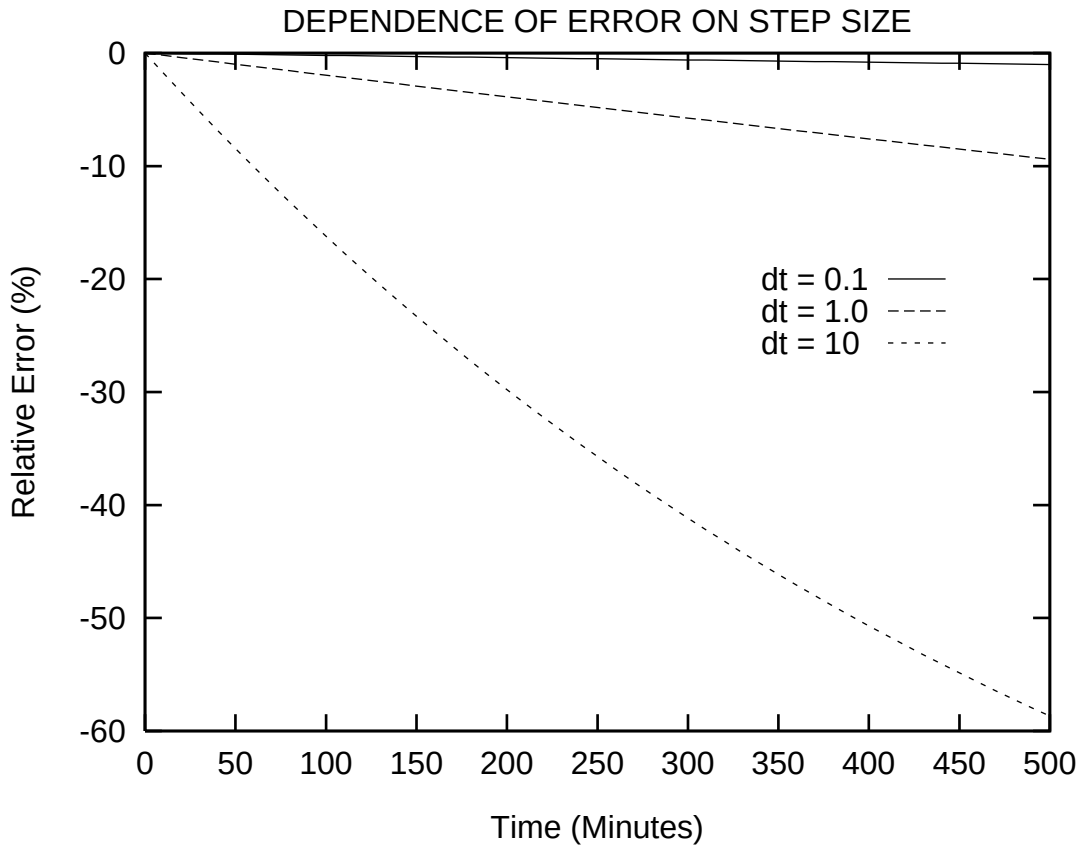

Figure 13−4. Comparison of the dependence of the relative error obtained using the method in Section 13−1 for three time steps. The values of the parameters were $n_0 = 1000$, and $T_b = 50$.

13.2.4 Initial Conditions

The unwanted generosity of the differential equation in giving us more solutions than we sought is a general feature of differential equations. The problem of finding which of the solutions is the one we want is called an *initial value* problem if the independent variable is time, and a *boundary value* problem if the independent variable is a spatial dimension. In general, an n^{th} order differential equation will give a solution with n unknown constants which must be found by providing n additional pieces of information about the solution. In our case, the differential equation we were trying to solve was first order, so there was one unknown constant which could be determined by specifying the number of bacteria $t = 0$. As another example, in the last chapter the height of the projectile, y, was the solution of a second order differential equation, Eq. (12−5), so there were two unknown constants involved in finding the solution. The two additional pieces of information about the solution I used to find them were the height of the projectile above the river at $t = 0$, and the speed of the projectile at $t = 0$. I took the initial height to be 0, and the initial speed to be 1000 m/sec. That information was sufficient to determine the solution completely.

The initial value problem appears in the numerical solution of differential equations also. Differential equations are usually solved numerically by using the given equation to obtain a formula which, given the value of the unknown variable at one time, provides approximately the value of the variable at a slightly later time. For example in our bacteria population problem, if we know the value of n at the earliest time for which we want to solve Eq. (13−5), then with the formula in Eq. (13−6) we can get approximately the value of n for any later time by applying the formula sequentially. With a little luck, the whole thing works pretty well, except that to start the chain off we need a value of n. That's the initial value problem rearing its head.

13.2.5 For the Skeptics Among You

Before continuing, let me try to reassure those of you who think I've pulled a fast one with my guessing of the solution. You may be thinking, "What if we had guessed some other functional form? Wouldn't we have then derived a different solution to the equation?" That shouldn't happen, because the colony should do whatever it does, independent of what some crazy guy like myself writes down on a piece of paper. If all of this has any value, what I write down on paper ought to match the behavior of the colony. Under a given set of circumstances the colony does only one thing, so I should not be able to find more than one solution of the governing equation which satisfies the given initial condition. What happens is that if we had guessed some other functional form we would have been unable to satisfy the differential equation, Eq. (13−5).

To illustrate, let's try my second guess, a power of t, and see if I can make it work. I should fail miserably. If I guess $n(t) \stackrel{?}{=} At^p$, where A and p are constants, I see right away that this won't work because I can't even satisfy the initial condition. With the guessed solution at $t=0$ we must have $n(0) = A\,0^p = 0$, whereas the initial condition requires that $n(0) = n_0$, and n_0 is probably not zero. I've got a brown belt in algebra, so it takes more than a little detail like that to whip me! I'll try instead

$$n(t) \stackrel{?}{=} A(t + t_0)^p \tag{13−15}$$

where t_0, A, and p are constants to be determined so that Eq. (13−15) is a solution of Eq. (13−5), and that the initial condition is satisfied. Doing the latter first (that's where I just stepped in it above),

$$n(0) = n_0 \stackrel{?}{=} A(0 + t_0)^p = At_0^p$$

so

$$t_0 = \left(\frac{n_0}{A}\right)^{\frac{1}{p}}$$

So far, so good. Plugging Eq. (13−15) into Eq. (13−5) yields

$$Ap\,(t + t_0)^{p-1} = \frac{A}{T_b}(t + t_0)^p \tag{13−16}$$

We can divide both sides by A and $(t + t_0)^{p-1}$, giving

$$p = \frac{t + t_0}{T_b} \tag{13−17}$$

This equation requires p to be a function of t. But by assumption, p was a constant, so I can find no solution of the form of Eq. (13−15) with constant p and my second guess did not work.

SURGEON GENERAL'S WARNING:
The following paragraph is not recommended for mathaphobics.
It has been known to cause severe headache, dyspepsea, and athlete's foot.

Maybe you are saying, "Why be such a martinet? So p isn't a constant, big deal. Who, other than you, said it had to be constant anyway?" Well if so you are right. We can guess whatever we want, and we could guess a solution of the form of Eq. (13−15) with p a function of t. Taking that trail leads straight through a swamp, but I didn't spend $14.95 for my brown belt for nothing! I definitely do *not* recommend solving the equation this way but I'm trying to convince you that the guessing method isn't as flaky as you

may think. Things get algebraically more complicated because the derivative $\frac{dn}{dt}$ is not so simple (we have to be very careful about using the chain rule), and because we end up with a differential equation for p. On the off chance you go in for such sadistic rituals, here is what you get. (There are a lot of intermediate steps I've left out.)

$$\frac{dp}{dt} + \frac{p}{(t+t_0)\log(t+t_0)} = \frac{1}{T_b \log(t+t_0)}$$

That differential equation can be solved, and after some slogging we find (*not* easily!)

$$p = \frac{B + t/T_b}{\log(t+t_0)}$$

where B is a constant to be determined and log means the natural logarithm. Putting that into Eq. (13−15) gives after some more slogging

$$n(t) = e^B e^{t/T_b}$$

Requiring $n(0) = n_0$ makes $B = \log(n_0)$ or $e^B = n_0$, and

$$n(t) = n_0 e^{t/T_b}$$

Slogging through the swamp got me finally the same answer I obtained with my original guess. I don't recommend doing things this way, but it can be done, and the results are consistent!

13.3 Numerical Solution of Differential Equations

Even before the advent of electronic digital computers, difficult differential equations were solved using numerical methods. The motions of the planets were worked out during the 1800's to good precision by solving Newton's Law using numerical methods and hand calculation. I understand that in the 1940's a room filled with people, each with a mechanical calculator, was used as a computer to solve some of the differential equations involved in designing the atomic bomb. With the advent of modern digital computers numerical solution of differential equations has become quite common. My goal in this section is to introduce you to some of the ideas behind the numerical solution of differential equations, and, as usual, to try to solidify your understanding of differential equations in general by presenting the material to you from a different viewpoint than you will see it in math classes.

I'll develop two related methods of solving Eq. (13−5) numerically. To save you the trouble of thumbing back to find it, the differential equation of interest is

$$\frac{dn}{dt} = \frac{n}{T_b} \tag{13−5}$$

where T_b is a constant that depends on the the private lives of the bacteria. We wish to solve this differential equation subject to some initial condition on n. As a start, I'll derive the same numerical method for solving Eq. 13−5 as I dreamed up in Section 13−1, except this time I'll do it from a different point of view. Looking at the weaknesses of this method suggests a better one, and we consider that method in the second subsection.

13.3.1 Euler's Method

Here's the idea behind the first method. It's the same idea I presented at the start of this chapter, but looked at from a different point of view. I'll consider only time points evenly spaced in time, with an interval Δt as in Fig. 13−5, and I'll approximate the derivative in Eq. (13−5) above by the forward-time approximation,

$$\left.\frac{dn}{dt}\right|_{t_i} \approx \frac{n_{i+1} - n_i}{\Delta t} \tag{13−18}$$

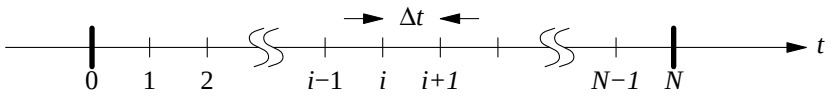

Figure 13−5. Schematic drawing showing the division of the time axis into equally spaced intervals.

Putting this in Eq. (13−5)

$$\frac{n_{i+1} - n_i}{\Delta t} \approx \frac{n_i}{T_b} \tag{13−19}$$

or, solving for n_{i+1},

$$\begin{aligned} n_{i+1} &\approx n_i + \frac{1}{T_b}\, n_i\, \Delta t \\ &= n_i \left(1 + \frac{\Delta t}{T_b}\right) \end{aligned} \tag{13−20}$$

This is the same formula as we obtained previously from a more physical viewpoint. The error with the method is shown in Fig. 13−3 for three time steps. For any time step the relative error keeps increasing with time. Why does it do that? The reason is that at any time, t, we estimate the value of n at the next time, $t + \Delta t$, using the slope at time t. The slope at all later times in the interval is larger than this value, but we make the estimate as if it remained at this minimum value throughout. Thus our approximation underestimates the value at $t + \Delta t$. This wouldn't be so bad because the error is small, but this underestimated value at $t + \Delta t$ means that when we take the next step, we start off with too small a slope. The slope we use for this next interval is even smaller than the correct value at its start. That leads to an even worse error at the end of the interval, which leads to a worse error in estimating the slope in the next interval, and so forth. This is a case of "the poor get poorer."

13.3.2 Modified Euler's Method

One obvious improvement on Euler's method is to use a numerical approximation that involves for each interval some kind of average slope throughout the interval. There are several possibilities for doing that. One of these leads to the popular *modified Euler's method*. I'll discuss this method here, and leave another of the possibilities to you to work out in an exercise at the end of the chapter.

Here is the idea behind the modified Euler's method. The problem with the original Euler's method is that it approximates the slope throughout a timestep by the slope at the start of the timestep. Better would be to use the slope halfway through. In terms of this slope, a better approximation to $n(t+\Delta t)$ than that used for Euler's method would be

$$n(t+\Delta t) \approx n(t) + \left.\frac{dn}{dt}\right|_{t_{i+\frac{1}{2}}} \Delta t \tag{13−21}$$

where $t_{i+\frac{1}{2}}$ is shorthand for $t_i + \frac{1}{2}\Delta t$. To use this formula we need a value for the derivative on the right hand side. That's easy to get—almost. The differential equation is

$$\frac{dn}{dt} = \frac{n(t)}{T_b} \tag{13−5}$$

This gives a formula for the slope at any time in terms of the value of n at that time. Therefore, the slope at $t_{i+\frac{1}{2}}$ is just

$$\left.\frac{dn}{dt}\right|_{t_{i+\frac{1}{2}}} = \frac{n(t_{i+\frac{1}{2}})}{T_b} \tag{13−22}$$

Putting this in Eq. (13−21) gives

$$n(t+\Delta t) \approx n(t) + \frac{n(t_{i+\frac{1}{2}})}{T_b}\,\Delta t \tag{13−23}$$

There's only one little rub. What the heck is $n(t_{i+\frac{1}{2}})$? There are several ways to obtain an approximation. The one that leads to the modified Euler's method uses the regular Euler's method to advance the solution at $n(t_i)$ by a half time step to $n(t_{i+\frac{1}{2}})$. Thus we write

$$n(t_{i+\frac{1}{2}}) \approx n(t_i) + \frac{n(t_i)}{T_b}\,\tfrac{1}{2}\,\Delta t \tag{13−24}$$

That does it. To march a solution forward by one time step from $n(t_i)$ to $n(t_{i+1})$ using the modified Euler's method, we first apply Eq. (13−24) to obtain $n(t_{i+\frac{1}{2}})$, and then we use this value in Eq. (13−23) to obtain $n(t_i+\Delta t)$.

The modified Euler's method can be implemented easily on a worksheet. Before doing so, let's march a few steps by hand just to make sure the method is clear. As previously, I'll consider the problem for which $n(0)=1000$ and $T_b = 50$ min, and I'll use a time step of $\Delta t = 10$ min. Then to obtain $n(10)$ we have to first estimate $n(5)$ using Eq. (13−24).

$$n(5) \approx n(0) + \frac{n(0)}{50} \times \tfrac{1}{2} \times 10$$

$$= n(0)\cdot\left(1 + \frac{10}{100}\right)$$

$$= 1000 \cdot 1.1 = 1100$$

Now we use this value in Eq. (13−23) to obtain $n(10)$,

$$n(10) \approx n(0) + \frac{n(5)}{50} \times 10$$

$$= 1000 + \frac{1100}{5} = 1220$$

I'll do one more time step. To calculate $n(20)$, I will need $n(15)$.

$$n(15) \approx n(10) + \frac{n(10)}{50} \times \tfrac{1}{2} \times 10$$

$$= n(10)\cdot 1.2 = 1220 \cdot 1.2 = 1464$$

and

$$n(20) \approx n(10) + \frac{n(15)}{50} \times 10$$

$$= 1220 + 1464 \times \frac{1}{5} = 1512.8$$

That's as far as I want to go manually. To implement the modified Euler method, I started to modify the Quattro Pro worksheet I had already made, but decided it was better to start over. You may prefer to modify your old worksheet. I'll tell you what I did, but I suggest you work it out yourself before looking

very closely at my solution. In any case, I suggest saving the old worksheet for future reference.

For easy comparison of the Euler and modified Euler methods, I put both on the same spreadsheet. I put the time in column A, and the modified Euler and regular Euler results for the population at that time in columns C and D. I put the exact solution in column E, and the errors in columns F and G. To march forward one time step with the modified Euler method I first need to march forward half a time step using Euler's method, so I put the Euler approximation to the population at the half time step in column B. One point that gave me a little trouble was deciding which row to use for the estimate of $n_{i+\frac{1}{2}}$. Should it go in the row for time t_i or for t_{i+1}? I chose to put it in the row for t_i. That way, to calculate the modified Euler value of $n(t_{i+1})$ I use values of n in the t_i row.

Here are the details of what I did. In cells A7..C7 I put the values of the parameters n_0, T_b, and Δt. For the half-step Euler calculation I needed $1 + \frac{1}{2}\Delta t/T_b$, and I put that value in E7. For the modified Euler, full step calculation I needed $\Delta t/T_b$, and put that in F7. For the regular, full step Euler calculation (included here only to facilitate comparison of the two methods) I needed $1 + \Delta t/T_b$, and I put that value in G7. To help me, I put appropriate titles above each of these cells, and labeled the columns.

I started the table of values for n in row 12. For the time column I put 0 and +A12+$C$7 in A12 and A13 respectively. For the half-step Euler column I put $A$7*$E$7 in B12 and +D13*$E$7 in B13. For the full-step, modified Euler column I put $A$7 and +C12+B12*$F$7 in C12 and C13. For the full-step regular Euler column I put $A$7 and +D12*$G$7 in D12 and D13. Finally, for the exact result column, I put $A$7*@EXP(A12/$B$7) in E12, and copied that into E13. To make comparison of the accuracy of the two methods easier, I put the relative error of both in columns F and G. For the modified Euler error column I put the formula (C12-E12)/E12 in F12, and copied it into F13. For the regular Euler column, the formula was (D12-E12)/E12. Finally, I was ready to finish by copying A13..G13 down about 200 rows. Fig. 13−6 shows the first few rows of my spreadsheet.

The results were rather impressive. For $\Delta t = 10$ min and $T_b = 50$ min, the modified Euler method gave results more than 10 times more accurate than did Euler's method. The variation of the error with step size, Δt, is of interest, and Fig. 13−7 shows the errors obtained for three values of Δt. Notice that the data are displayed using a logarithmic scale because of the wide range of values. The figure shows that the error scales nearly as the square of Δt. (If Δt is made 10 times smaller, the error decreases by a factor of about 100.) This behavior marks the modified Euler's method as being of second order. For comparison, we found the regular Euler's method to be only first order.

13.4 An Example from Electrical Engineering, a RC Circuit

Let's look at a circuit analysis problem in which a simple differential equation arises. I'll first solve it analytically, and then numerically. Consider the circuit in Fig. 13−8a. It consists of a capacitor connected in parallel with a resistor, and the pair are connected to a voltage source through a switch. The switch has been closed for a long while, and then at some time, which we call $t = 0$ for convenience, the switch is opened so that the combination is no longer connected to the battery. The question is "What is the voltage, v, across the resistor (and also the capacitor) for all times?".

For now, I'll leave the values of the voltage source, V, the capacitance, C, and the resistance, R, unspecified, and develop equations which contain these symbols. My reasons for doing so are the same ones I had for leaving parameters unspecified in the bacteria calculation. First it's easy to apply the equations I develop to circuits with differing values of these parameters. If we considered a circuit with specific values, such as say $V = 5$ volts, $C = 100\ \mu$F, and $R = 30$ kΩ, then the numbers would soon get all mixed up, and we'd have to do the analysis all over again for a circuit with different values. Second, at each stage of the calculation it's easier to see how the value of each component enters into each equation. That gives better insight into how the circuit behaves, and it makes it easier to spot some kinds of mistakes. Generally, even if I only want to find the current or voltage in one specific circuit with specific values of the components, I try to do most of the calculation using algebraic symbols for the values and substitute numbers in for them only at the end.

a)

	A	B	C	D	E	F	G
1	Chapter 13: Comparison of the Euler's and modified Euler's						
2	methods for calculating the bacteria population						
3							
4							
5	PARAMETERS				CONSTANTS		
6	No.	Tb	dt		1+*dt/2Tb	dt/Tb	1+dt/Tb
7	1000	50	1		1+0.5*C7/B7	+C7/B7	1+C7/B7
8							
9							
10		Euler	n(t)			Error	
11	t	n(t+1/2	Mod Eul	Reg Eul	Exact	Mod Eul	Reg Eul
12	0	+$A$7*$E$7	$A$7	+$A$7	+$A$7*@EXP(A12/$B$7)	(C12-E12)/E12	(D12-E12)/E12
13	+A12+$C$7	+D13*$E$7	+C12+B12*$F$7	+D12*$G$7	+$A$7*@EXP(A13/$B$7)	(C13-E13)/E13	(D13-E13)/E13

b)

	A	B	C	D	E	F	G
1	Chapter 13: Comparison of the Euler's and modified Euler's						
2	methods for calculating the bacteria population						
3							
4							
5	PARAMETERS				CONSTANTS		
6	No.	Tb	dt		1+*dt/2Tb	dt/Tb	1+dt/Tb
7	1000	50	1		1.01	0.02	1.02
8							
9							
10		Euler	n(t)			Error	
11	t	n(t+1/2)	Mod Eul	Reg Eul	Exact	Mod Eul	Reg Eul
12	0	1010	1000	1000	1000	0	0
13	1	1030.2	1020.2	1020	1020.201	-1.3E-06	-0.0002
14	2	1050.804	1040.804	1040.4	1040.811	-6.5E-06	-0.00039
15	3	1071.82	1061.82	1061.208	1061.837	-1.6E-05	-0.00059
16	4	1093.256	1083.256	1082.432	1083.287	-2.8E-05	-0.00079
17	5	1115.122	1105.122	1104.081	1105.171	-4.5E-05	-0.00099
18	6	1137.424	1127.424	1126.162	1127.497	-6.5E-05	-0.00118

Figure 13−6. The first few rows of the modified Euler method worksheet for calculating the bacteria population. Part a) shows the formulas in each cell, and b) shows the worksheet as it appears on the screen.

Back to the analysis, we need an equation involving v. To get it we choose a likely looking node and require the net current into the node to equal the net current out of it. With any luck, expressing these currents in terms of voltages (via the current-voltage relationships for the individual components) will give us the equation we desire. Let's pick the node labeled a in Fig. 13−8b and define currents i_S, i_C, and i_R, as shown. Then, setting the current leaving the labeled node equal to the current entering it,

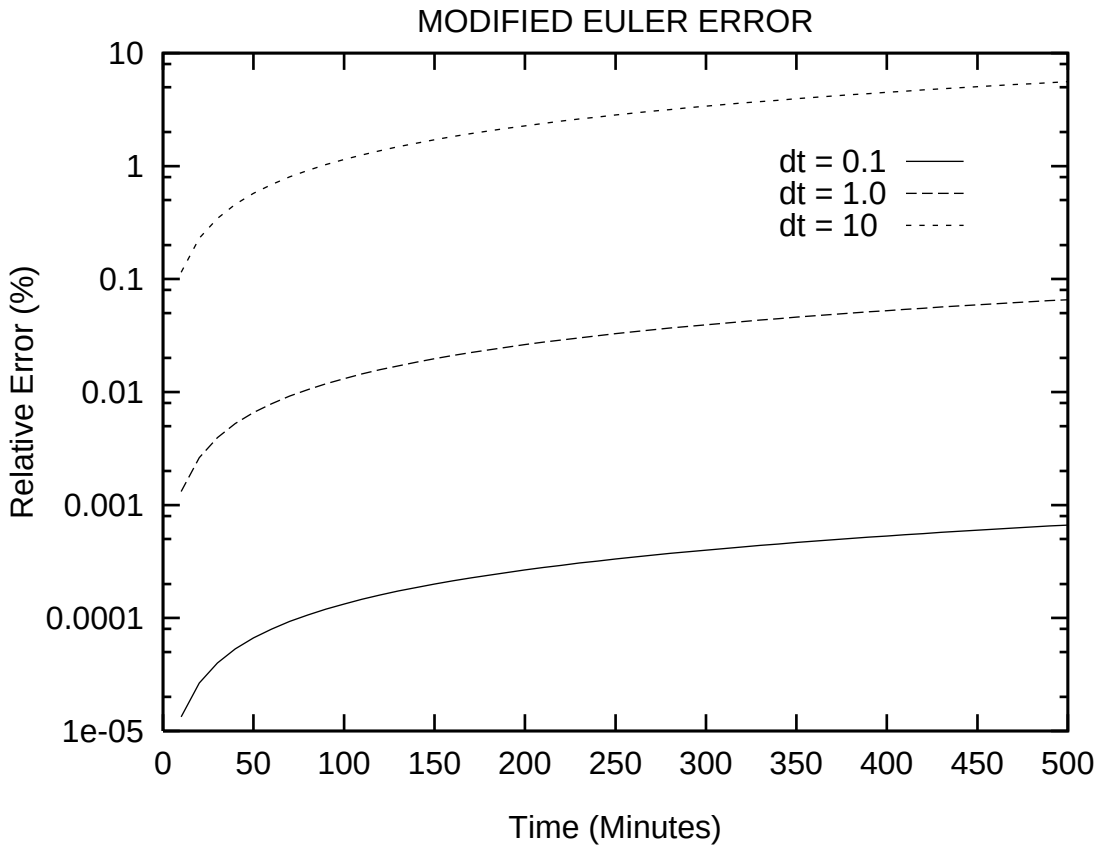

Figure 13−7. Comparison of the relative error for three step sizes using the modified Euler's method. The step sizes are shown, and the values of the parameters are the same as those used for Fig. 13−4. Because of the logarithmic scale used, the absolute value of the relative error is plotted.

$$i_C + i_R = i_S \tag{13−25}$$

That is one equation with three unknowns, so we still have some work to do. Each of the currents depends on the voltage, v, so I will use these current-voltage relationships to try to get an equation involving v only. From Eq. (13−2),

$$i_C = C\,\frac{dv}{dt} \tag{13−26}$$

and from Ohm's law

$$i_R = \frac{v}{R} \tag{13−27}$$

Then Eq. (13−25) becomes

$$C\,\frac{dv}{dt} + \frac{v}{R} = i_S \tag{13−28}$$

Eq. (13−28) is a linear (thank goodness), first-order differential equation, with unknown dependent variable v, and independent variable t. The *dependent* and *independent* stuff just means that the dependent variable, v in this case, depends on the value of the independent variable, t in this case. There is also a second unknown dependent variable, i_S, which looks troublesome, but turns out not to be a problem.

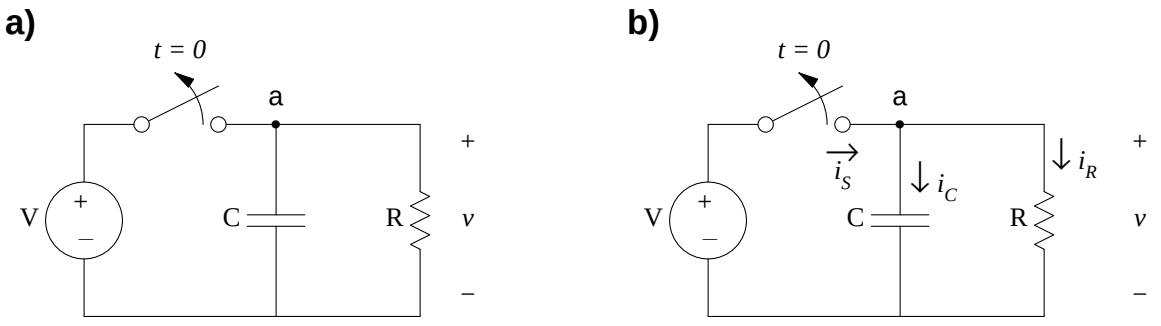

Figure 13−8. a) Circuit diagram for the RC circuit discussed in the text. The switch was closed for a long time, and was opened at $t = 0$. b) Same circuit as in part a), except the the currents in the three branches connected to node a are labeled.

When we say we want to solve Eq. (13−28), we mean we want to find a relationship or table giving the value of v for all allowed values of t. The solution is not a single number, as with an ordinary algebraic equation, but rather a whole table of numbers—a function. If we are solving the equation analytically we usually find an equation, rather than a table, relating v and t. If we are solving it numerically we usually calculate a table that gives the value of v for a number of discrete values of t. For the analytic solution the equation is like a table which has an infinite number of entries, one for each possible value of t. For the numerical solution, we cannot generate an infinite number of table entries, so we try to calculate entries for values of t sufficiently closely spaced as to give us whatever information we need about the solution.

Solving Eq. (13−28) for half of the possible times is very easy. For times before the switch is opened (in other words, $t < 0$), the top node is connected to the voltage source. Therefore, for these times

$$v(t) = V \qquad \text{for } t < 0 \tag{13−29}$$

a constant. As a side issue, you might want to know what i_S is during these times. If v is constant in time, then its derivative is zero, and Eq. (13−28) becomes

$$0 + \frac{V}{R} = i_S \qquad \text{for } t < 0$$

or

$$i_S = \frac{V}{R} \qquad \text{for } t < 0 \tag{13−30}$$

This result makes sense. Because the voltage across the capacitor doesn't change during this time, the current through it is zero, and it might as well not be there. The current through the resistor is, by Ohm's law, just V/R, and since no current flows into the capacitor this is just i_S, as Eq. (13−30) claims.

What about the other half of the times, those for which $t \geq 0$? The switch is opened at $t = 0$, so during these times it is open, and no current can flow through it. Therefore, $i_S = 0$ during these times, and Eq. (13−28) becomes

$$C\frac{dv}{dt} + \frac{v}{R} = 0 \qquad \text{for } t \geq 0 \tag{13−31}$$

This is a first-order, linear, homogeneous differential equation with one unknown dependent variable, v. The equation is of the same form as the differential equation we encountered for the bacteria population, and the same methods of solution will work for it. I'll solve the equation three times: once analytically, once using Euler's method, and once using the modified Euler method.

13.4.1 The Analytic Solution

First, I'll solve the differential equation analytically. Taking my own advice, I'll first guess an exponential and see if that works.

$$v(t) \stackrel{?}{=} a\, e^{bt} \qquad \text{for } t \geq 0 \tag{13−32}$$

where a and b are constants which we must find. Notice that, as before, by guessing a solution of this form with several unknown parameters in it I have reduced the level of clairvoyance required. I only have to correctly guess the right functional form (an exponential in this case), not the details (the values of the constants). I'll let the algebra tell me what these constants, a and b, must be if Eq. (13−32) is to satisfy Eq. (13−31).

Let's see how good my guess is. Plugging Eq. (13−31) into Eq. (13−30)

$$Cb\, a\, e^{bt} + \frac{a\, e^{bt}}{R} = 0 \qquad \text{for } t \geq 0 \tag{13−33}$$

In this equation $a\, e^{bt}$ divides out, and we obtain

$$Cb + \frac{1}{R} = 0 \qquad \text{for } t \geq 0 \tag{13−34}$$

or

$$b = -\frac{1}{RC} \tag{13−35}$$

If we choose this value for b, then Eq. (13−32) will be a solution of Eq. (13−31) for any value of a. Just as with the bacteria population example, we set out to find one solution to a differential equation but found an infinite number, one solution for each value possible for a.

Out of all the solutions to Eq. (13−31) I have just found, only one corresponds to the behavior of my circuit, and I am still left with the problem of finding out which one it is. What I need is to find what value of a corresponds to my particular problem. This is the initial value problem we encountered before. To solve the problem, I need one additional bit of information about the voltage at any time for which the differential equation describes the behavior of the circuit. What information have we not supplied to the math that should be important? Looking at Eq. (13−31) we see that there is no mention of the fact that the top node had been connected to a voltage source of strength V before $t = 0$ when the switch was opened. The voltage of the node should be different if the node was connected to a 1 volt source than say a 5000 volt source. That ought to be important, and it seems likely that it is the missing bit of information. In fact, each of the solutions I've generated describes the behavior of the circuit for some voltage source. There's a different solution for each possible value of the voltage of the source. The differential equation didn't know what source I had in mind, so it gave me all the solutions. My remaining task is to find the solution corresponding to the voltage source in my circuit.

To do that, we can notice that at $t = 0$, just when the switch is opened, the voltage across the capacitor must be the same as it was just before the switch opened. (This is not as obvious as it may sound. I discuss the subject in a little more detail in Section 13−5.4.) Thus we require

$$v(0) = V \tag{13−36}$$

Using this requirement with Eq. (13−32), we have

$$v(0) = a\, e^{0} = a = V \tag{13−37}$$

That's the desired value of a, and the voltage for all times greater than 0 is

$$v(t) = Ve^{-t/RC} \qquad \text{for } t \geq 0 \tag{13−38}$$

Let's look at this a little closer. For $t = 0$, $v(0)$ is $Ve^0 = V$, as expected. (Remember that $e^0 = 1$.) For $t > 0$, we would expect that the voltage on the capacitor would fall off as it discharges through the resistor. This is the behavior predicted by Eq. (13−38) because for positive t the argument of the exponential is negative, and the argument becomes more and more negative as t increases, causing v to become smaller and smaller. How long does it take the capacitor to discharge? Well it never completely discharges, there is always a *little bit* of voltage left on it, but that little bit becomes very little indeed. When $t = RC$ the argument of the exponential is –1, v is $v(RC) = Ve^{-1} = V/e$, and the value has fallen to $1/e \approx 0.368$ of the original value. This time is often taken as a measure of the discharge time, and is called the *fall time*, and the Greek letter tau, τ, is often used for it.

As an aside, there was another way we could have found the analytic solution of Eq. 13−31. Defining $\tau = RC$, and putting the second term in the equation on the other side of the equals sign gives

$$\frac{dv}{dt} = -\frac{v}{\tau} \tag{13−39}$$

This is almost the same equation that the bacteria population, n satisfied in Eq. 13−5). Besides the change of variable from n to v, the only difference is that this equation has $-\tau$ in the denominator on the right side instead of T_b. Looking at the solution we obtained earlier, we could substitute $-\tau$ for T_b, obtaining

$$v(t) = a\, e^{t/(-\tau)} = a\, e^{-t/RC} \qquad \text{for } t \geq 0 \tag{13−38}$$

13.4.2 The Numerical Solution

As with the analytic solution, we first notice that the unknown, v, obeys a different equation before the switch is opened than after, so the solution must be carried out in two regimes: $t < 0$, and $t \geq 0$. The equation v satisfies in the first is $v(t) = V$. That's trivial to solve, and we certainly don't need a numerical algorithm for it. For $t \geq 0$, v satisfies Eq. (13−31). Proceeding as before, the Euler's method "marching" formula is

$$v_{i+1} \approx v_i \left(1 - \frac{\Delta t}{RC}\right) \qquad \text{Euler} \tag{13−40}$$

and the modified Euler's method marching formula is

$$v_{i+1} \approx v_i - v_{i+\frac{1}{2}} \frac{\Delta t}{RC} \qquad \text{Modified Euler}$$

where

(13−41b)

$$v_{i+\frac{1}{2}} = v_i \left(1 - \frac{\Delta t}{2RC}\right) \qquad \text{Modified Euler}$$

These formulae are very similar to those we developed for the bacteria population problem, and I will leave the implementation of them to you as an exercise.

13.5 A Second Example: An RLC Circuit

I'd like to consider as a second example the circuit shown in Fig. 13−9. The switch has been closed for a very long time, and then is opened at $t = 0$. The question is "What is the voltage, $v(t)$, for all times?". As in the previous example, we will try to answer the question by writing down an equation expressing the fact that the net current into the labeled node must equal the current out of it, and then using the current-voltage relationships for the individual components connected to the node to convert the equation to one involving the node voltage. We will find that everything proceeds more or less as before, except for the complication that we end up with two coupled differential equations rather than just one. If we were trying to solve the problem analytically that would be a significant complication, but we will find that it only adds a little more algebra in the numerical case.

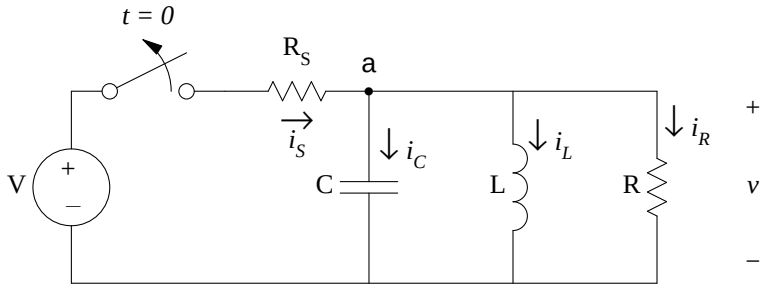

Figure 13−9. Schematic diagram showing the RLC circuit discussed in the text. The switch was closed for a long time before being opened at $t = 0$. The currents in the four branches connected to node a are labeled.

To get started, we'll label all the currents as shown in Fig. 13−9. As in the previous example, we have two time eras to consider: $t < 0$, when the switch is closed, and $t \geq 0$, when the switch is open.

13.5.1 The t < 0 Era

First we'll consider the era $t < 0$. The problem isn't quite as simple as previously, because the voltage source is connected to the node of interest through a resistor labeled R_S, so we can't just say that v is V. The voltage across the inductor (which happens to be the same as v) is, according to Eq. (13−3)

$$v_L = v = L\,\frac{di_L}{dt}$$

I don't know right now what i_L is but the switch has been closed for a long time, and i_L should have stopped changing. It should have become constant, meaning that the derivative should have become zero. Therefore

$$v(t) = 0 \qquad \text{for } t < 0 \tag{13−42}$$

You may be uncertain about my claim that i_L must be a constant for $t < 0$. In one of the problems at the end of this chapter, you are asked to find v and i_L for the problem in which the switch has been open for a long time so all voltages and currents in the RLC part are zero, and then at $t = 0$ the switch is closed. As long as the resistor, R, is not infinite you will find that the voltages and currents change quite a bit at first, but become more and more constant with time. For right now, if you are skeptical, "trust me." You'll have an opportunity to check it out later.

What are all the currents for $t < 0$? If $v = 0$, then both i_C and i_R must also be zero because $i_C = C\,\frac{dv}{dt}$, and $i_R = \frac{v}{R}$. Also, the voltage across R_S is $V - v$, which must be just V since $v = 0$, so

$$i_S(t) = \frac{V}{R_S} \qquad \text{for } t < 0 \tag{13−43}$$

The current into the labeled node is i_S, and the current out of it is $i_C + i_L + i_R = i_L$, so i_L must equal i_S,

$$i_L(t) = \frac{V}{R_S} \qquad \text{for } t < 0 \tag{13−44}$$

13.5.2 The t ≥ 0 Era

That pretty much takes care of the $t < 0$ era, what about $t \geq 0$? For these times the switch is open, so $i_S = 0$, and the no-net-current law applied to node a gives

$$i_C + i_L + i_R = 0 \qquad \text{for } t \geq 0 \tag{13−45}$$

I would like to convert this to an equation for v. Both i_C and i_R have simple, convenient relationships with v, but the current-voltage relation for the inductor is backwards from the way I would like it:

$$v = L\,\frac{di_L}{dt} \tag{13−46}$$

There are several ways around this problem. One would involve integrating both sides of the above equation to get a relationship giving i_L in terms of an integral of v. Another way would be to differentiate both sides of Eq. (13−45). I'll take a different path. I'll just leave i_L alone, and deal with two unknown dependent variables, v of course, and i_L. Since I have two unknowns, I'll need two equations. One will be Eq. (13−46) above, and the other will be Eq. (13−45) with the current-voltage relationships substituted in for i_C and i_R. Using $i_C = C\,\frac{dv}{dt}$ and $i_R = \frac{v}{R}$ in Eq. (13−45) we get

$$C\,\frac{dv}{dt} + \frac{v}{R} + i_L = 0 \qquad \text{for } t \ge 0 \tag{13−47}$$

These two equations, Eqs. (13−46) and (13−47) are our two equations for the two unknowns, v and i_L. For convenience I'll rearrange them so that the derivatives are on the left hand side, and the rest of the stuff is on the right.

$$\begin{aligned} \frac{di_L}{dt} &= \frac{v}{L} \qquad \text{for } t \ge 0 \\ \frac{dv}{dt} &= -\frac{i_L}{C} - \frac{v}{RC} \qquad \text{for } t \ge 0 \end{aligned} \tag{13−48b}$$

To solve these equations numerically, we could use either Euler's method or the modified Euler's method. I'll choose the latter. The process goes just as before, except there are two, coupled, differential equations instead of just one.

13.5.3 The Numerical Method

As before I'll choose a time step, Δt, and calculate approximate values for the unknowns at a set of discrete times, $\{t_k\}$, separated by the time step, Δt. Notice I've changed the subscript from i as in the last section to k to avoid confusion with the currents. I also ran into a problem with too many subscripts on i_L, so I decided to drop temporarily the L subscript. For the rest of this section, i and i_L will by synonymous. The derivatives evaluated halfway between these times are

$$\begin{aligned} \left.\frac{di}{dt}\right|_{k+\frac{1}{2}} &\approx \frac{i_{k+1} - i_k}{\Delta t} \\ \left.\frac{dv}{dt}\right|_{k+\frac{1}{2}} &\approx \frac{v_{k+1} - v_k}{\Delta t} \end{aligned} \tag{13−49b}$$

For the values of the derivatives I'll use Eqs. (13−48)

$$\begin{aligned} \left.\frac{di}{dt}\right|_{k+\frac{1}{2}} &= \frac{v_{k+\frac{1}{2}}}{L} \\ \left.\frac{dv}{dt}\right|_{k+\frac{1}{2}} &= -\frac{i_{k+\frac{1}{2}}}{C} - \frac{v_{k+\frac{1}{2}}}{RC} \end{aligned} \tag{13−50b}$$

Equating the half-interval derivatives in the corresponding equations yields

$$i_{k+1} \approx i_k + \frac{v_{k+\frac{1}{2}}}{L}\Delta t$$
$$v_{k+1} \approx v_k - \left(\frac{i_{k+\frac{1}{2}}}{C} + \frac{v_{k+\frac{1}{2}}}{RC}\right)\Delta t \qquad (13\text{−}51b)$$

As before, these equations are great but we need to know v and i at the half-interval times. To get those, we use Euler's method with a time step of $\frac{1}{2}\Delta t$.

$$i_{k+\frac{1}{2}} \approx i_k + \frac{v_k}{L}\frac{\Delta t}{2}$$
$$v_{k+\frac{1}{2}} \approx v_k - \left(\frac{i_k}{C} + \frac{v_k}{RC}\right)\frac{\Delta t}{2} \qquad (13\text{−}52b)$$

These two equations combined with Eqs. (13−51) are sufficient to let us march along.

13.5.4 The Initial Value Problem

To start the march we need the initial values of v and i. (Remember that I'm using the symbol i in place of i_L, the current through the inductor.) Just before the switch was opened, v was zero, and i was V/R_S. I claim that these values must be the same just after the switch opened, so $v(0)=0$, and $i(0) = V/R_S$. That claim is not as obvious as it might sound. It's based on two old adages:

1. "The voltage across a capacitor cannot change abruptly;" and
2. "The current through an inductor cannot change abruptly."

These adages in turn are based on the current-voltage characteristics of capacitors and inductors. For a capacitor, for example,

$$i_C = C\frac{dv_C}{dt}$$

If v_C were to change suddenly from one value to another, the derivative would be quite large (even infinite if the change occurred instantaneously). A very large or infinite derivative then implies (unless $C=0$) a very large or infinite current through the capacitor. Therefore, in the absence of such huge capacitor currents, the capacitor voltage must change continuously. Similar arguments applied to the current-voltage relation for an inductor imply that sudden changes in the current through an inductor require infinite voltage across it.

Anyway, our initial conditions are

$$i(0) = \frac{V}{R_S} \qquad (13\text{−}53a)$$

and

$$v(0) = 0 \qquad (13\text{−}53b)$$

13.5.5 Programming Quattro-Pro

We are now ready to program Quattro Pro, and see how this all works. Quattro Pro works with numbers, not algebraic symbols, so we have to choose specific values for the components. To start, I chose R=1 MΩ (=10^6 Ω), L=0.1 H, C=1 μF (=10^{-6} F), R_S = 5 kΩ, and V = 5 volt. Here's how I did it. I suggest you do the programming on your own. I put the values of t for which I will calculate i and v in column A, the values of i and v at the half-intervals in columns B and C respectively, and the values of i and v at the whole intervals in columns D and E. I started the table in row 15 to give room for a title, the adjustable parameters, and some constants. In cell A7 I put the value of V, and in cells B7..D7 I put the

values of R, L, and C, respectively. In `E7` and `F7` I put R_S, and the time step, Δt. The initial conditions depend of some of these parameters, and I put them in a separate box below, with i_0 in `B11` and v_0 in `C11`.

To make the table, I put `0` in cell `A15`, `+$B$11` in `D15` and `+$C$11` in `E15`. Adopting the same convention I used previously, I decided to put the values of $i_{k+\frac{1}{2}}$) and $v_{k+\frac{1}{2}}$ in the t_k row rather than the t_{k+1} row. I put `+D15+E15*$F$7/(2*$C$7)` in `B15`, `+E15-(D15+E15/$B$7)*$F$7/(2*$D$7)` in `C15`, and I copied these formulae into `B16..C16`. In `A16` I put `+A15+$F$7`. I put `+D15+C15*$F$7/$C$7` in `D16` and `+E15-(B15+C15/$B$7)*$F$7/$D$7` in `E16`. Finally, I copied `A16..E16` down the worksheet for about 500 rows. With that many numbers to calculate, the worksheet gets a little sluggish. Fig. 13−10 shows the first few rows of my worksheet.

The result is two columns of numbers in columns `D` and `E` which aren't very informative. In this case it really is worthwhile to graph the results. Plotting either v or i, you should find what look like sine waves. In fact they almost are sine waves. If the resistor labeled R weren't there (or it had infinite resistance) they would be exactly sine waves. The presence of the resistor causes the amplitude of the sine waves to decrease slowly with time. The rate of decrease is determined by the value of R, with smaller values causing a faster decrease. You can see this effect by decreasing the value of R in cell `C3` and regraphing. Be sure to wait for Quattro Pro to finish recalculating (as evidenced by the disappearance of the `BKGD` at the bottom-center of the screen). For small enough values of R, the damping of the sine wave becomes so great that it never gets to complete a full cycle. (If you decrease R too much, the numerical method becomes unstable and starts giving ridiculous results. This is a common problem when solving differential equations numerically, and I discuss it in a little more detail in Section 13−5.7.)

The oscillatory behavior of the circuit for large values of R is a common feature of circuits containing both inductors and capacitors, and the general phenomenon is called *resonance*. The phenomenon is very important in electronic circuit engineering. The voltage oscillates with a frequency that is determined mostly by the values of L and C and to a lesser extent by R. The situation is closely analogous to the vibration of a string. If you pull on some part of a guitar string, for example, that is analogous to connecting the RLC circuit to a steady voltage source, and letting go of the string is analogous to opening the switch in the circuit, disconnecting the circuit from the source. The response of the two systems is similar; both oscillate or ring at a set frequency, the resonant frequency. In music the phenomenon is useful for generating audible tones of specific frequency, and in electrical circuit design it is useful for generating a sinusoidal voltage waveform of a specific frequency.

Very early radio transmitters, in fact, consisted only of an inductor and capacitor, a battery, and a rotary switch to alternately connect and disconnect the LC circuit with the battery. The LC circuit would ring at some frequency determined by the component values each time the connection was made or broken, and the switch had to be rotated fast enough that the switch would close and open again before the ringing waveform decayed too far. These days, more sophisticated circuits using vacuum tubes or transistors for the old mechanical switch are used, but the idea is nearly the same.

13.5.6 A Synopsis of the Analytic Solution

The set of differential equations Eqs. (13−48) can be solved analytically, but we won't do it here. Instead I'll just give you a brief synopsis of how it goes. It turns out that my advice about guessing an exponential works, but there's this little detail about the argument of the exponential which takes a lot more explaining than I want to do here. First, we guess

$$i_L(t) = i_0\, e^{st} \qquad \text{and} \qquad v(t) = v_0\, e^{st} \tag{13−54}$$

where i_0, v_0, and s are unknown constants similar to a and b in the previous example. Notice that I've become a little more sophisticated in using my guessing technique. I recognized that if I can find such solutions the constants before the exponentials will be the values of i_L and v at time $t = 0$, and gave these constants more descriptive names than the a I used previously. Also, I called the constant that appears in the argument to the exponential s to conform to common usage. Plugging these guesses into Eqs. (13−48) we find that they will work if

a)

	A	B	C	D	E	F
1	Chapter 13: Numerical solution of RLC circuit using					
2	the modified Euler's method.					
3						
4						
5	PARAMETERS					
6	V	R	L	C	Rs	dt
7	5	50	0.1	1E-06	5000	1E-05
8						
9	CONSTANTS					
10	i0	v0				
11	+A7/E7	0				
12						
13		Euler		Mod Euler		
14	t	i(t+1/2)	v(t+1/2)	i(t)	v(t)	
15	0	+D15+E15*$F$7/(2*$C$7)	+E15-(D15+E15/$B$7)*$F$7/(2*$D$7)	+$B$11	+$C$11	
16	+A15+$F$7	+D16+E16*$F$7/(2*$C$7)	+E16-(D16+E16/$B$7)*$F$7/(2*$D$7)	+D15+C15*$F$7/$C$7	+E15-(B15+C15/$B$7)*$F$7/$D$7	

b)

	A	B	C	D	E	F
1	Chapter 13: Numerical solution of RLC circuit using					
2	the modified Euler's method.					
3						
4						
5	PARAMETERS					
6	V	R	L	C	Rs	dt
7	5	50	0.1	1E-06	5000	1E-05
8						
9	CONSTANTS					
10	i0	v0				
11	0.001	0				
12						
13		Euler		Mod Euler		
14	t	i(t+1/2)	v(t+1/2)	i(t)	v(t)	
15	0	0.001	-0.005	0.001	0	
16	1E-05	0.000999	-0.0131	0.001	-0.009	
17	2E-05	0.000997	-0.01972	0.000998	-0.01637	
18	3E-05	0.000995	-0.02514	0.000996	-0.0224	
19	4E-05	0.000992	-0.02956	0.000994	-0.02732	
20	5E-05	0.000989	-0.03315	0.000991	-0.03133	

Figure 13−10. Worksheet which uses the modified Euler's method to solve for the voltage and current in Fig. 13−9. Part a) shows the formulas in each cell, and b) shows the worksheet as it appears on the screen.

$$s = -\frac{1}{2RC} \pm \frac{1}{C}\sqrt{\frac{1}{4R^2} - \frac{C}{L}} \tag{13−55}$$

As usual, i_0 and v_0 can be anything. There is potentially some funny business with the square root. If R is small enough, there is no problem because the first term in the argument of the square root will be larger

than the second, and the difference will be positive. If R is made bigger, however, there will come a point where the second term exceeds the first, and the argument of the square root becomes negative. All "normal" numbers when squared give a positive number, so what is meant by the square root of a negative number?

Most of you have seen such a situation before. To handle square roots of negative numbers a new type of number was invented—an *imaginary* number. A "normal" number is called a *real* number, and a number consisting of the sum of a real and an imaginary number is called *complex*. All that is OK I guess, s just becomes complex when R exceeds some value. The problem is that s is used in an exponent, and raising e to some complex power seems a bit questionable at best. This is the part I don't want to discuss here, so I'll just tell you that e to a complex power can be defined so as to be sensible. I'm leaving some stuff out here, but the result is that e raised to an imaginary power varies with the magnitude of that power like a sine wave. You already know that e raised to a real power just increases or decreases monotonically, depending on the sign of the number.

Thus as R is varied, from very large values to very small, there ought to be a distinct change in behavior as R crosses through the value for which the argument of the square root changes sign. Specifically, we expect oscillatory solutions for values of R above this "critical" value, and simple exponentials for values less. The critical value is the one which makes the argument of the square root zero,

$$R_{crit} = \frac{1}{2}\sqrt{\frac{L}{C}} \tag{13−56}$$

If R just equals R_{crit}, the circuit is said to be *critically-damped*. If R is smaller than this value it is *over-damped*, and if R is larger, the circuit is *under-damped*. Although the exact point of transition from under-damped to over-damped is not clearly evident, you should find this behavior if you experiment with the worksheet we have just created. Values of R much larger than R_{crit} produce damped oscillatory solutions, values smaller produce simple decaying exponentials. Fig. 13−11 shows the results for three values of R. The value of R_{crit} for the circuit values used is about 158 Ω.

You will derive all this stuff analytically when you take EE213. Many students find all the calculus and algebra a bit rough going. I suggest you save this program on a floppy, put it somewhere, and drag it out again when you get to this point in EE213. I think you will find it helpful.

13.5.7 Numerical Instability

I can also use this example to demonstrate a common problem with numerical solution of differential equations—instability. You may have inadvertently encountered it already. If you make R too small, the worksheet bombs out. To see that, replace all parameters to their default values ($C = 1$ μF, $L = 0.1$ H, and $\Delta t = 10^{-5}$ sec), except set R equal to 1 Ω. You should find that the worksheet claims that both the voltage and current rapidly become very large. That doesn't make sense. For example, at $t = 0.0002$ sec, the worksheet claims the voltage has risen to more than 10^{29} volts and the current to more than 10^{25} amp! If you keep everything the same, except reduce Δt to 10^{-6} sec, then everything looks reasonable again. Obviously, something is behaving strangely.

Here's what's going on. As the value of R is decreased, the voltage changes more and more rapidly just after $t = 0$ (check it out for yourself). The approximations in Eqs. (13−49) are based on the assumption that the quantity involved doesn't change too much during a time interval. This assumption becomes less and less accurate as R is made smaller with everything else held constant. If we go too far, the solution becomes unstable in that the errors start increasing exponentially. For a given value of R, the problem can be fixed by choosing Δt sufficiently small that the change in v during any time interval is small enough to keep everything stable. Alternately, for given values of R and Δt for which the solution is well behaved, it is possible to find a new value of Δt large enough to make the solution go unstable. In fact if you reset the worksheet parameters to the original values, $R = 1$ MΩ, $C = 1$ μF, and $L = 0.1$ H, but choose a little larger value of Δt, like 5×10^{-5} sec, it displays an interesting behavior.

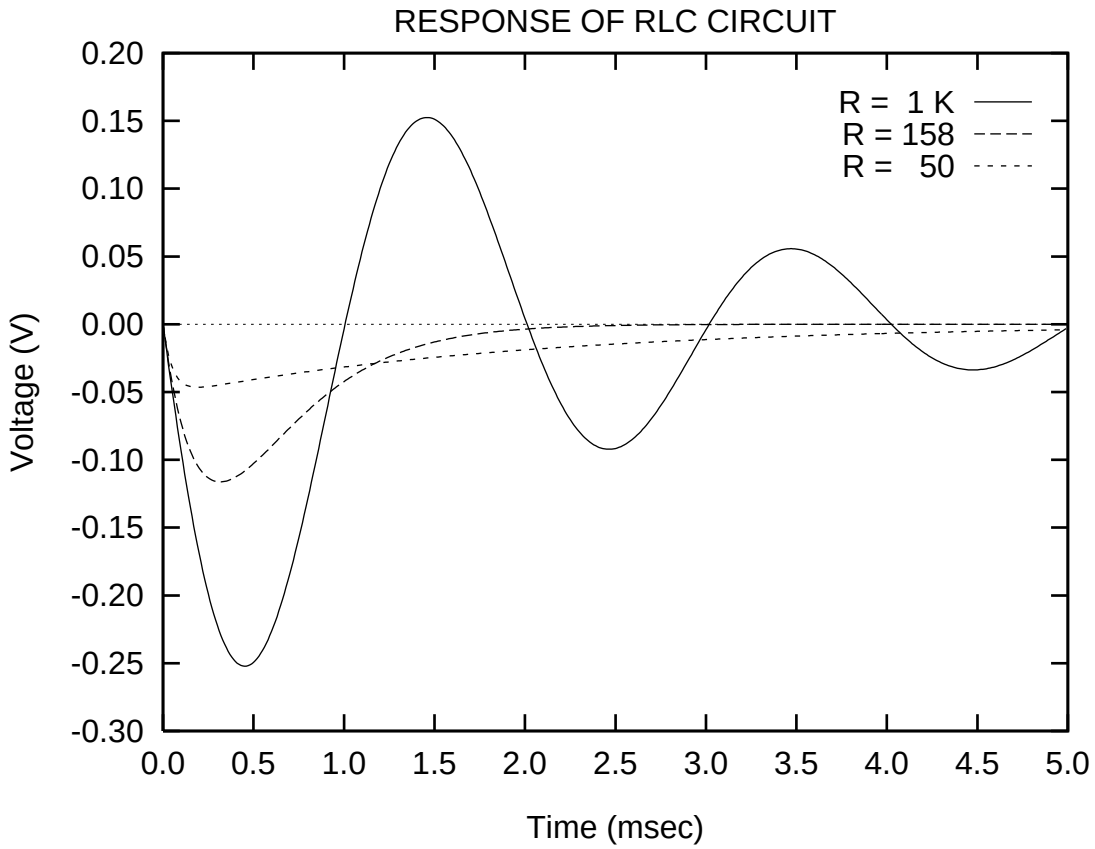

Figure 13−11. Voltage v in the circuit in Fig. 13−9 for three values of R. The first value is much larger than R_{crit}, the second is about equal to R_{crit}, and the third is much less.

This problem of the stability of numerical methods in general is a difficult one. For problems as simple as this one, it is not very hard to determine stability criteria for the standard numerical methods such as the modified Euler method we used. For more complex problems, such as those involving non-linear differential equations, the question of the stability of any particular method can be much harder to determine. In multi-dimensional problems where the dependent variable is a function of several independent variables (like x, y, and z for example) some methods which look very reasonable turn out to be unconditionally unstable! We aren't going to pursue the subject any farther here. In deciding whether to apply numerical analysis to a problem, my advice is to try it, and do a reality check on the answer. If it looks reasonable, it's probably OK. Numerologists consider it heresy, but a pretty good check on the accuracy of your implementation is to halve the time step (or whatever), and see how much the answer changes. It isn't foolproof, but the largest change in your answer is often a pretty good estimate of how accurately you solved the differential equation.

13.6 Exercises

1. Modify the model of the bacteria colony to apply to mortal bacteria. Assume that the number of bacteria that die per unit time is given by $n(t)/T_d$, where T_d is the average life span of a bacterium.

   a. Obtain an exact, analytic solution for $n(t)$ and then use the modified Euler's method to find a numerical solution. Use the same values for $n(0)$ and T_b as used in Section 13−1, and take $T_d = 200$ min. Compare the analytic and numerical results.

   b. What happens if the death rate exceeds the birth rate? For example, what do the numerical and analytic solutions predict for the population function if $T_d = 20$ min?

   c. Suppose the birth rate exactly equals the death rate. What ought to happen then? Do your analytic and numerical solutions agree?

2. Program the problem discussed in Section 13−4 to find the voltage as a function of time for the RC circuit shown in Fig. 13−8. Use both Euler's method and the modified Euler's method, and compare the results of both with the exact, analytic solution. Verify that the error scales with Δt as expected.

3. Consider the following differential equation.

$$\frac{dy}{dt} + \frac{1}{t}\,y = 0 \tag{13−57}$$

   subject to the initial condition $y(1) = 3$.

   a. Solve the equation analytically by taking my advice in the notes on guessing solutions.

   b. Program a Quattro Pro work sheet that solves Eq. (13−57) for $t \geq 1$ using both the Euler's method and the modified Euler's method.

   c. Calculate the error with the numerical methods for several, suitably-chosen values of the time grid spacing, Δt. Do the errors of the two methods vary with Δt as expected?

4. Consider the differential equation

$$\frac{dy}{dt} - y^2 = 1 \tag{13−58}$$

   subject to the initial condition $y(0) = 0$.

   a. Verify that neither of my recommended guesses for the solution of this equation work.

   b. Program Quattro Pro to solve the differential equation using the modified Euler's method. Calculate a solution for $0 \leq t \leq 1$.

   c. The solution of Eq. (13−58) is known to be $y = \tan t$, where tan means the tangent, and the argument is assumed to be in radians. Use this solution to calculate the error with the numerical solution for several values of Δt.

   d. The analytic solution has strange behavior for certain values of t. How does the numerical solution behave as t approaches the smallest of these?

5. Consider the following three differential equations. For which can one find a solution of the form $y = Ae^{bx}$ where A and b are constants and $A \neq 0$? WHY?

   a. $\dfrac{d^2y}{dx^2} + \dfrac{dy}{dx} - 6y = 0$

   b. $\dfrac{d^2y}{dx^2} + x\,\dfrac{dy}{dx} - 6y = 0$

c. $\dfrac{d^2y}{dx^2} + 2(x+1)\dfrac{dy}{dx} + 4xy = 0$ [Careful!]

6. Consider the following differential equation

$$\frac{dy}{dt} + t^2 y = \sin \omega t$$

where ω is a known constant, and $y(0) = 3$.

a. Develop a set of formulas based on the modified Euler's method to solve the differential equation numerically using a time increment Δt.

b. Estimate the largest value of Δt which you could use and still expect reasonable accuracy. Explain your reasoning.

c. Take $\omega = 1000$, and implement your formulas using Quattro Pro. Determine the solution throughout at least one cycle of the sin function. Compare the solution you get for several values of Δt. How good was your estimate in the previous part?

7. The goal of this problem is to verify the discussion leading up to the determination of the initial conditions for the circuit in Fig. 13−9. (See the three paragraphs just before Eq. (13−44).) Consider the circuit in Fig. 13−9, but with the modification that the switch has been open for a long time, and then is closed suddenly at $t = 0$. Use the following values for the components, $R = 10$ kΩ, $L = 0.1$ H, $C = 1$ μF, $V = 5$ volt, and $R_S = 5$ kΩ.

a. What are the values of $v(t)$ and $i_L(t)$ before the switch is closed ($t < 0$)? (This part's easy!)

b. Make use of the two adages to determine the initial conditions, $v(0)$ and $i_L(0)$.

c. Program Quattro Pro to find $v(t)$ and $i_L(t)$ for times after the switch is closed ($t \geq 0$). Use the modified Euler method as in the notes.

d. Carry out you calculation far enough to see if i_L and v approach after a "long" time the same values claimed in the notes.

8. Here is the idea behind another modification to Euler's method for solving first order differential equations numerically. Consider the solution of Eq. (13−5),

$$\frac{dn}{dt} = \frac{n}{T_b} \tag{13−5}$$

As in the derivation of the modified Euler's method, we use the derivative at time $t_{i+\frac{1}{2}}$ to advance the solution from time t_i to t_{i+1}, to get Eq. (13−24).

$$n_{i+1} \approx n_i + n_{i+\frac{1}{2}} \frac{\Delta t}{T_b} \tag{13−24}$$

As before, we have the problem of finding a suitable value for the half-time-step value, $n_{i+\frac{1}{2}}$. In the modified Euler derivation, we estimated this value using Euler's method and a time step of $\frac{1}{2}\Delta t$. Another possibility would be to approximate $n_{i+\frac{1}{2}}$ by the average of the values of n at t_i and t_{i+1},

$$n_{i+\frac{1}{2}} \approx \tfrac{1}{2}(n_i + n_{i+1})$$

Show that this idea leads to the marching formula

$$n_{i+1} \approx n_i \cdot \frac{1 + \frac{1}{2}\Delta t / T_b}{1 - \frac{1}{2}\Delta t / T_b}$$

and program a Quattro Pro worksheet to implement the method. Compare the behavior of the

method with that of the modified Euler's method, using the same values for T_b and n_0 as in the notes.

9. The two methods discussed in the text for solving differential equations numerically are directly applicable only to first order differential equations. They can be used, however, to solve second and higher order differential equations through the use of a clever trick. Consider, for example, the second order differential equation

$$\frac{d^2y}{dt^2} + f(t)\frac{dy}{dt} + g(t)y = h(t) \tag{13−59}$$

where $f(t)$, $g(t)$, and $h(t)$ are some given functions. The trick is to define a new variable, w, by

$$w = \frac{dy}{dt} \tag{13−60}$$

We can put this equation in Eq. (13−59) to reduce it to a first order differential equation, but now it involves two unknown variables, y, and w. With two unknowns we need a second equation, and Eq. (13−60) suffices admirably because it is also only first order. Summarizing, we have

$$\frac{dw}{dt} = h(t) - f(t)w - g(t)y \tag{13−61a}$$

and

$$\frac{dy}{dt} = w \tag{13−61b}$$

Solving these coupled equations analytically could present a problem, but they can be solved numerically without difficulty using the same technique used in the notes to find the voltage and inductor current for Fig. 13−9. We have two first order differential equations, so we will need two initial conditions, one on y, and the other on w (which is just $\frac{dy}{dt}$).

Use this idea to solve numerically the harmonic oscillator equation,

$$\frac{d^2y}{dt^2} + \omega^2 y = 0 \tag{13−62}$$

where ω is a constant. Solve the equation for $\omega = 1000$, and for initial conditions $y(0) = 1$ and $y'(0) = 0$. (y' is the first derivative of y.) Graph your result. Solve for whatever ranges of t you like, but turn in a worksheet with a solution covering the range $0 \le t \le 0.02$.

10. Another, more straight-forward, method for solving second-order differential equations involves using the results of Exercise 12−4 for a centered approximation to the second derivative of a function evaluated at a grid point. You should have found

$$\left.\frac{d^2f}{dt^2}\right|_i \approx \frac{f_{i+1} + f_{i-1} - 2f_i}{\Delta t^2} \tag{13−63}$$

This is a centered approximation and it should be accurate to second order. Eq. (13−63) can be used to obtain an approximate marching formula, which allows you to find an approximation to f_{i+1}, given that you know values of f_i and f_{i-1}.

Apply such a method to the numerical solution of Eq. (13−62) with the same value of ω, and for the same initial conditions. To get started, you will need values of $y_0 = y(0)$ and $y_1 = y(\Delta t)$. The initial condition provides a value for y_0, but y_1 is a little harder. One way around the problem is to estimate y_1 using the same idea as behind Euler's method and the given value of $y'(0)$. Use that idea or any other you can think of to carry out the numerical solution, and compare your results with those obtained in the previous problem. For this equation, the method should be accurate to second

order. Is it?

11.

a. Use the method discussed in Exercise 13−7 to solve numerically the differential equation in the previous chapter, Eq. (12−5), that describes the motion of the projectile. Use the same values of the parameters and the same initial conditions as used there. Compare your result with the exact, analytic solution given in Eq. (12−4).

b. The guessing method almost works to find an analytic solution to Eq. (12−5). Notice first that the equation reduces to a first order differential equation by making the substitution

$$f = \frac{dy}{dt} \tag{13−64}$$

The equation becomes

$$\frac{df}{dt} = -\frac{\gamma}{m} f - g$$

This equation is of the same form as Eq. 13−5 except for the addition of a constant, $-g$, to the right side. The guess of an exponential doesn't quite work because of this constant, but the following guess does

$$f \stackrel{?}{=} a\, e^{bt} + c$$

where a, b, and c are constants to be determined so that the guess solves the differential equation and any applicable initial conditions. Knowing f, we can integrate it, according to Eq. (13−64) to obtain y. Show that the correct analytic solution is, in fact, given by Eq. (12−4).

12. Bessel's equation is a famous differential equation for which there is no solution in terms of ordinary functions. The equation appears in a wide range of engineering fields, including such diverse applications as the frequency bandwidth for frequency-modulated (FM) signals, and the behavior of circular waveguides. Perhaps a more familiar example is the peculiar sound of a drum. If the drum head is circular, the vibration of the drumhead is described by Bessel's equation. Bessel's equation of zeroth order is

$$\frac{d^2y}{dt^2} + \frac{1}{t}\frac{dy}{dt} + y = 0$$

Use the method discussed in Exercise 13−7 above to solve numerically this equation subject to the initial conditions

$$y(0) = 1 \qquad\qquad y'(0) = 0$$

Graph your result for $0 \le t \le 8$.

This solution has been studied extensively, and is commonly denoted by the symbol $J_0(t)$. You can find tables giving the values of J_0 for a wide range of values of t, and there are a number of formulas that have been developed for the derivative and various integrals involving J_0. So you can check your answer the Table 13−1 gives the values of J_0 for several values of t.

BESSEL FUNCTION	
t	**J_0(t)**
0.0	1.0000
1.0	0.7652
2.0	0.2239
3.0	-0.2601
4.0	-0.3971
5.0	-0.1776
6.0	0.1506
7.0	0.3001
8.0	0.1717

TABLE 13−1. Table of values of $J_0(t)$, the ordinary Bessel function of the first kind of zeroth order.

14. COMPUTER ARCHITECTURE

In this chapter, we will look at just what's going on inside a typical computer. We discussed this subject very briefly in Chapter 1, and this would be a good time to review that material. Here I want to go into considerably more detail.

As shown in Fig. 1-1, the central processing unit, or CPU is at the center of the computer. This is a complicated circuit which carries out the tasks given the computer. Generally several types of memory are connected to the CPU. This memory contains the program the computer is to execute, and the data the computer is to operate on. Most data and programs are stored permanently on either a floppy disk or a hard disk. There are usually also several types of devices such as a keyboard, and a monitor connected to the CPU to allow the operator to input information into the computer, and to obtain information from it. I'll discuss the memory and the CPU in somewhat more detail separately.

14.1 Memory Architecture

Computers can hold a large volume of data in electronic memory. The computers in the lab can store a little more than four million bytes of data. In order to make use of this vast storage area, some scheme is needed to label each byte so that it can be accessed when desired. The scheme which is used involves assigning each byte a number, called an *address*, and the information stored in it is then accessed by using this address. The memory organization for a computer with a very small memory is shown schematically in Fig. 14−1.

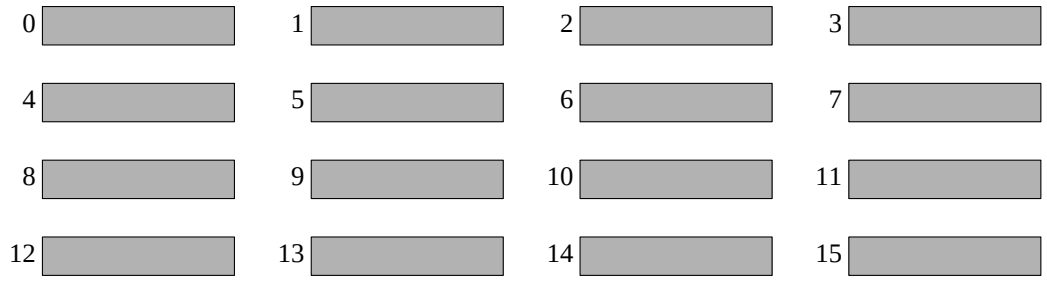

Figure 14−1. Schematic diagram of the memory organization in a very small computer. The one-byte

cells are shown as shaded rectangles, and the address of each cell is shown at the left. For your convenience each address in the diagram is expressed in decimal or base 10, but the computer uses binary.

A useful computer would require much more memory. Both program instructions and data to be operated on are stored in the memory (in different locations of course).

14.2 The CPU

The CPU is made up of a number of sub-circuits, each with a specialized job. These days, the sub-circuits are all fabricated on one or sometimes a few small pieces of silicon, and the CPU is called a *microprocessor*. You can buy such microprocessors for prices ranging from a few dollars to several hundred dollars. The microprocessor used in the computers in the lab is made by Intel, and has the number 80386. The CPU has a repertoire that consists of a set of basic instructions which it knows how to carry out. For example, some instructions transfer a binary number from memory to a storage area inside the CPU, or vice versa; others carry out arithmetic operations such as the addition of two numbers; and others compare two numbers to determine which is larger. Each instruction consists of a binary number called the *op code*, possibly followed by one or more additional numbers called *operands*. The op code tells the CPU what is to be done and the *operands* tell the CPU what it is to be done to. The Intel 80386 has an instruction set consisting of 152 different instructions.

In operation, the CPU executes sequentially a set of instructions, called a program. One of the specialized sub-circuits inside the CPU is the *instruction decoder*. The instruction to be executed is loaded into this circuit, and it figures out just what is to be done and sends the appropriate signals to other sub-circuits within the CPU to do it. Remember that each instruction is simply a number (either 1 or 2 bytes long on the 80386), so the instruction decoder must contain circuitry to identify each possible instruction number with an action, determine from the instruction whether or not it is supposed to be followed by operands, read in the operands if they are supposed to be there, and then finally turn on or off the relevant sub-circuits in the correct order and with the correct timing to carry out the requested operation. The instruction decoder is not a simple circuit!

Some of the specialized sub-circuits in the CPU serve as places where numbers can be stored temporarily for use by the CPU. These sub-circuits are called *registers*. The number of bits a register can hold depends on the particular type of CPU, and may also depend on which register it is. Most of the registers in the 80386 microprocessor hold 32 bits. There are 8 general purpose, 32-bit registers, and two 32-bit special-purpose registers associated with the operation of the CPU. There are also 6 16-bit special-purpose registers called *segmentation registers* which are used for dividing memory up into smaller chunks. The use of these latter segmentation registers is somewhat confusing, and you will not need to know much about them for this course, so I'll just ignore them here. In case you are curious or masochistic, I discuss the use of segmentation registers along with several other distasteful features of the 80386 microprocessor briefly in the appendix to this chapter.

The general purpose registers are used for temporary storage of data while it is being manipulated. For example, to add two numbers located in memory one might copy each number into a different internal register and add the contents of the two registers, storing the result in a third register. This result could then be copied back to some location in memory, or used in some subsequent calculation. More commonly, the registers are used to store intermediate results. For example, the CPU has an instruction to add two integers, but not three or more. To add three integers stored in memory, one might first add two of them, storing this intermediate result in a register, and then add the third number to the contents of this register. Since the intermediate result is of no interest, it need not be preserved by writing it to memory. Moving data to or from memory takes time, so one generally tries to do as much of the manipulation as possible using data already contained in a CPU register.

One specialized register is called the *program counter*, or *instruction pointer*, and its purpose is to keep track of which instruction is to be executed next. A program to be executed is generally stored

sequentially in memory with the first instruction at the start of the memory block, then the second instruction, then the third, and so forth. To execute the program, the memory address containing the first instruction is loaded into the program counter, and the CPU is told to start. It retrieves the instruction located at the address in the program counter (corresponding to the first instruction in the program) and places it in the instruction decoder. The instruction decoder adds the length in bytes of the instruction to the contents of the program counter, and proceeds to carry out the instruction. When it has finished, the CPU repeats the process, loading the instruction starting at the address in the program counter (corresponding now to the second instruction) into the instruction decoder, and so forth. The process continues until the CPU encounters an instruction to halt, or an instruction it can't decode.

14.3 Communication between the CPU and External Circuits

Perhaps at this point you are wondering how the CPU communicates with memory or other devices. The details for the 80386 are complicated, and I don't want to get into all of them here, so the following is an abridged description. It's correct as far as it goes, but several details are omitted.

There are two sets of pins on the microprocessor intended for connection to external circuits. One set is for address information, and the other is for data information. Each of these sets of pins are connected to corresponding sets of wires which in turn are connected to several other circuits in the computer. Such a set of wires is called a *bus*. The set which carries address information is called the *address bus*, and the set carrying data information is called the *data bus*. When the microprocessor places a number on either bus, all circuits connected to that bus receive it. For example, all memory circuits receive an address when it is placed on the address bus. The memory circuitry is designed, however, so that only the cell corresponding to the address on the bus responds. If the CPU is trying to read the information in that cell, the cell responds by placing the number it contains on the data bus. It knows the CPU wanted to read a number rather than write one because there is another, one-wire, bus connected to all memory cells and the CPU which specifies whether a read or a write operation is desired. All the other cells are also connected to the data and read/write buses, but they do not respond because they were not addressed. The CPU can then read the number on the data bus with confidence that it is the number stored in the addressed cell.

Similarly, if the CPU wants to write a number into a specific memory cell, it places that number on the data bus, places the address of the desired cell on the address bus, and finally sets the read/write bus line to write. The addressed cell responds by storing the number on the data bus in itself. When this happens, whatever number was previously stored in that cell is destroyed.

Peripheral devices such as the keyboard or display, or a disk are not connected directly to the CPU. Rather, each device is connected to its own interface circuit separate from the microprocessor. The interface circuits each contain several special-purpose internal registers, some of which are used to store data temporarily, and others to pass commands from the CPU to the device. These circuits are also connected to the data and address buses, and each is assigned unique addresses for its externally-accessible registers. The CPU communicates with a particular device by placing the address of the desired register of the device on the address bus, and then either reading from or writing to the associated register.

14.4 More About Instructions

Most of the instructions in the instruction set of a CPU fall into one of several groups. One group of instructions corresponds to copying data from one place into another. In the 80386, these are called *move* instructions. One can move a number from an address in memory to an internal register or vice versa; from one internal register to another; or from one memory location to another. Thus a move instruction might consist of a number specifying that the operation is a move, with a different number being used for each possible set of source and destination (memory-register, register-register, etc.), followed by two more numbers which specify which register or memory address is to be used for the source and destination, respectively.

Another group of instructions are the arithmetic operations. Most CPU's have instructions for integer

addition and subtraction, and many have multiply and divide instructions as well. Other arithmetic instructions include one to change the sign of a number, instructions to shift the bits either to the right or the left in a register, and instructions to set specific bits in a number to either 0 or 1. Most CPU's do not have instructions for doing floating point arithmetic, but many manufacturers of microprocessors do offer a companion, specialized integrated circuit, called a *floating point processor* or FPU, that does. The FPU is connected to the CPU, and has its own instruction set. When an FPU is connected, the CPU assumes that any instruction it does not recognize must be an instruction for the FPU and passes the instruction on. Floating point arithmetic is possible without an FPU, but you have to "fake it," by using integer arithmetic operations to manipulate floating point numbers. The 80386 microprocessor does not do floating point arithmetic, but Intel offers a companion floating point processor, the 80387.

Another group of instructions are the *compare* instructions. One can check to see if the contents of some location is greater or less than zero, if it's zero or non-zero, or if the contents of one location is larger than, smaller than, equal, or unequal to the contents of another. Another group of instructions, some of which are used in conjunction with the compare instructions, are the *jump* instructions. These instructions modify the program counter so that the next instruction executed is one other than the one following the current instruction. These instructions are used frequently to control flow in the program. For example, if the result of some calculation is zero one might want to execute one small set of instructions, and to execute another set otherwise. To do so, one could use the appropriate compare instruction, followed by a jump on zero instruction.

14.5 Programming the CPU

The program that the CPU sees consists of a sequential set of numbers (in binary of course). To write an executable program directly, you would have to have a list of instructions the CPU recognizes coupled with the corresponding binary code for each. You would have to figure out how the CPU could accomplish the task you have in mind using only the instructions in the list, write a program consisting of an ordered list of the instructions required to carry out the task, translate each instruction to the corresponding binary code, and then enter the resulting ordered list of numbers into memory. Even for CPU's with simple instruction sets, this job becomes formidable for all but the simplest programs. The difficulty is not so much deciding what operations the CPU should perform, but rather that of getting all the details exactly right. Every quantity which is to be stored in memory must be assigned a unique address, and that address must be used every time that quantity is to be accessed. If the CPU is to jump from one instruction to another, the jump instruction must include the exact address of the instruction to which the CPU is to jump. If, during program development, one or more instructions are inserted into the program, then the addresses of all the instructions after the inserted instructions change, and any references to these addresses in jump instructions must then also be changed. Details such as these really start to pile up, and writing programs in machine language becomes very tedious.

Today programs are rarely written in machine language directly, but in the early days of electronic computing, that was the only way to program computers. Once written on paper, the programs were entered into memory manually with a bank of switches (one switch for each bit). It was quickly realized that the part of programming that humans were the worst at (getting all the details right) was the kind of thing a computer is very good at. Programs, called *assemblers* were written which associated each instruction in the CPU's repertoire with a suggestive set of letters, called a *mnemonic*. The instruction to move data from one location to another, for example, might be given the mnemonic `MOV`, and the instruction to add two numbers might be given `ADD`. Internal registers were given names such as `R1`, `R2`, etc. Thus one might write an assembly language statement to move the contents of `R2` to the memory location (written here in hexadecimal) `1A3C` as `MOV 1A3CH,R2`, where the `H` in the address means the number is given in hexadecimal. This is an obvious improvement over the machine language version which might be (in hexadecimal) something like `89B41A3C`. This use of mnemonics relieves the programmer of the task of looking up or remembering the machine codes for each instruction, and it makes the program much easier for humans to read. The programmer still has to remember all the mnemonics, but that is a lot easier than a lot

of seemingly random numbers.

In these notes I use hexadecimal whenever I want to write the machine language version of an instruction, or an address. The reason for using hexadecimal is that it is easy to convert from binary to hex and vice versa, and it's easy to divide numbers expressed in hex into 8-bit bytes. Each hex digit corresponds to four bits, so two digits correspond to a byte. Octal can also be used for this purpose, but dividing a number expressed in octal up into bytes is not so easy because each octal digit corresponds to only three bits, rather than four. Two thirds of the third digit goes into the low byte, and one third into the high byte. The disadvantage of using hex is that you have to get used to dealing with 16 digits instead of just 8.

You may be wondering why I have not yet given an example of specific instructions for the 80386. The reason is that the 80386 is a real mess! Fortunately, you can remain blissfully ignorant of most of these details, and I show you just a little bit of the mess in an appendix in case you go in for such things.

14.6 A Fictional CPU with a Simple Instruction Set

I want to discuss some short examples of programming at both the assembly language and the machine language levels. Unfortunately the instruction set of the 80386 is complicated and messy to use, and even the simplest example of programming at either level would require much more explanation than I want to go into here. Other CPU's exist which are considerably simpler, but even they have complications in their instruction sets that I don't want to discuss. For this reason, I've made up my own, fictional CPU along with its instruction set. The CPU is a 16-bit machine, and I've called it the PFW007. It has four 16-bit, general purpose registers called `R0`, `R1`, `R2`, and `R3`, and three special purpose registers called `PC`, `CR`, and `SP`. The `PC` register is the program counter, and you do not have direct access to it. The `CR` register is called the *Condition Register* and it is used to implement conditional jumps. The use of `CR` is discussed later in this chapter. The `SP` register is called the stack pointer, and the use of it is discussed in the next chapter. The instruction set includes only 26 different instructions, and is shown in Table 14−1. (Many of these instructions are just minor variations of other instructions, so one might argue that there are really fewer than 26.) Several aspects of the table need explaining.

14.6.1 The Move Instructions

The largest group of instructions involves copying a number from one place to another. The `MOV` instruction copies the contents of the second operand into the first. The convention used with the assembly language mnemonics is that the first operand is the one that gets modified. (You might find it helpful to think of the comma and space between the two operands as an equal sign.) Nine forms of the `MOV` instruction are listed explicitly, and several other instructions such as `PUSH` and `POP` are really move instructions. The first `MOV` instructions does a register-to-register move. (You might think of it as `Ra = Rb`.) For example, suppose that register `R1` contains `36` and register `R3` contains `17`, then after executing a `MOV R1, R3` instruction both `R1` and `R3` would contain `17`, the value originally stored in `R3`. The second `MOV` instructions loads a 16-bit integer into a register, and the integer to be loaded is given explicitly as an operand. For example, after executing a `MOV R2, #100` instruction, register `R2` would contain the (hexadecimal) integer `100`.

The third form does a register-to-memory move, and the fourth a memory-to-register move. For these instructions the address in memory is given explicitly as an operand, and this form of addressing is called *direct addressing*. For example, if the memory location `300` contains `249`, then after executing `MOV R0, 300`, both memory address `300` and register `R0` would contain `249`. Notice the difference between the instructions `MOV R0, 0300` and `MOV R0, #0300`. The first copies the contents of the memory location `0300` into register `R0`, whereas the second puts the number `0300` itself into the register. The fifth and sixth forms also transfer a number between a register and memory, but in these cases, the memory address is taken to be the number stored in a register. This form of addressing is called *indirect addressing*. In the assembler mnemonic, I indicate this form of addressing with the `@` sign. Thus, if register `R1` contains `30A0` and `R2` contains `0012`, then the instruction `MOV @(R1), R2` would copy `0012` to memory location `30A0`. You might think of the at sign and parentheses as meaning something like "the

PFW007 INSTRUCTION SET

Assembler Mnemonic	Action	Machine Instruction	Length (bytes)
`MOV Ra, Rb`	Rb→Ra	$1(4\times R_a + R_b)$	1
`MOV Ra, #Num`	Num→Ra	$2(R_a)nnnn$	3
`MOV Adr, Rb`	Rb→Adr	$2(4 + R_b)aaaa$	3
`MOV Ra, Adr`	Adr→Ra	$2(8 + R_a)aaaa$	3
`MOV @(Ra), Rb`	Rb→Address in Ra	$3(4\times R_a + R_b)$	1
`MOV Ra, @(Rb)`	Contents of Adr in Rb→Ra	$4(4\times R_a + R_b)$	1
`JMP #Adr`	#Adr→PC	2C*aaaa*	3
`JSR #Adr`	Push PC, #Adr→PC	2D*aaaa*	3
`RTS`	Pop→PC	2F	1
`CMP Ra, Rb`	Set CR according to Ra − Rb	$C(4\times R_a + R_b)$	1
`BR Cond, #Adr`	#Adr→PC if Cond&CR=Cond	5(*Cond*)*aaaa*	3
`PUSH Ra`	SP→SP − 2, Ra→@(SP)	$6(R_a)$	1
`POP Ra`	@(SP)→Ra, SP→SP + 2	$6(4 + R_a)$	1
`MOV SP, Rb`	Rb→SP	$6(8 + R_b)$	1
`MOV Ra, SP`	SP→Ra	$6(C + R_a)$	1
`MOV SP, #Adr`	#Adr→SP	2E*aaaa*	1
`ADD Ra, Rb`	Ra + Rb→Ra	$7(4\times R_a + R_b)$	1
`INC Ra`	Ra + 1→Ra	$8(R_a)$	1
`DEC Ra`	Ra − 1→Ra	$8(4 + R_a)$	1
`LSHIFT Ra`	2Ra→Ra	$8(8 + R_a)$	1
`RSHIFT Ra`	Ra/2→Ra	$8(C + R_a)$	1
`SWAB Ra`	Swap bytes in Ra	$B(R_a)$	1
`AND Ra, Rb`	Ra & Rb→Ra	$9(4\times R_a + R_b)$	1
`OR Ra, Rb`	Ra | Rb→Ra	$A(4\times R_a + R_b)$	1
`NOT Ra`	˜Ra→Ra	$B(4 + R_a)$	1
`HALT`	Stop	00	1

TABLE 14−1. Table showing the instruction set of the fictional PFW007 CPU. `Ra` and `Rb` stand for registers, `Adr` for a memory address, and `Num` for a signed 16-bit integer. In the **Action** column, a register name or an address by itself should be interpreted to mean the contents of that register or address. A number or address with a pound sign, #, prepended should be interpreted to mean that number or address itself. The Machine Instruction and Length columns are explained in 14−6.5

address given by the contents of."

There is also a group of five move instructions in the group in the table headed by `PUSH Ra`. I could explain here what these instructions do, but I would have a difficult time explaining why anyone would want to use them. For that reason, I will delay discussion of these instructions to the next chapter, where I will be able to show you some examples of their use.

14.6.2 The Jump and Compare Instructions and the Condition Register

The next four instructions in the table cause the processor to jump to a specific instruction. Here I will only discuss `JMP` and `BR`. I will also discuss the fifth instruction in the group, `CMP`. As with the `PUSH` instruction, I could easily explain here what the `JSR` and `RTS` do, but it would be difficult to discuss how they are used. For that reason, I will delay discussing these two instructions until the next chapter when I

will be able to provide some specific examples.

The first instruction in this group, `JMP #Adr` is an unconditional jump. Following execution of this instruction, program execution continues with the instruction at the address provided as an operand. An equally sensible mnemonic for this instruction would have been `MOV PC, #Adr`. The instruction simply puts `Adr` into the program counter register, `PC`. The instruction `BR Cond, #Adr`, is a conditional jump, or `Br`anch. It's like the `JMP` instruction except that the jump will occur only if a specified condition is met. As with `JMP`, the address of the instruction to be optionally jumped to is given directly as an operand. This instruction is used to jump to differing blocks of code, depending on whether some value is zero, non-zero, negative, or positive; or on whether one value is equal to, unequal to, less than, or greater than another value.

The `BR` instruction operates in conjunction with a special purpose register called the *condition register* so that the jump only occurs if certain bits in the condition register are 1. The first (least significant) four bits are the only ones checked. These bits are set to either 0 or 1 every time a value is written to a register or memory, and every time the `CMP` instruction is executed. When a value is written to a register or memory location the first (least significant) bit is set to one only if the number stored was zero; the second only if the number stored was non-zero; the third if it was negative; and the fourth if it was positive. For example, if the number `0032` were stored the condition register would be `1010` in binary, or `AH` in hex, because the number was non-zero and positive. If the number were F321H, the condition register would be `0110`, or `6` in hex, because the number was negative and non-zero. Table 14−2 summarizes the effects on `CR`.

CONDITION REGISTER BITS				
Bit No. (CR value if set)	4 (8)	3 (4)	2 (2)	1 (1)
After writing a value	> 0	< 0	≠ 0	= 0
Executing `CMP Ra, Rb`	Ra > Rb	Ra < Rb	Ra ≠ Rb	Ra = Rb

TABLE 14−2. Effects of writing a value to a register or memory location, and of executing the `CMP` instruction on the condition register. Each of the four relevant bits of `CR` are shown in the table, and each bit is set to one if the condition shown in each entry is met and to zero otherwise. At the head of each column are given both the bit number and the value of the condition register if only this bit is set. The latter is the value in parentheses.

The `CMP` instruction is used to compare the size of two values. It sets the condition register without having any other effect, and it is intended to be used with the `BR` instruction. It really is not a jump instruction, but I included it with the jump group because of this close association with `BR`. When the `CMP Ra, Rb` instruction is executed, the first (least significant) bit of the condition register is set to one if the values in `Ra` and `Rb` are equal, the second if the two values were unequal, the third if the value in `Ra` was less than that in `Rb`, and the fourth if the value in `Ra` was less. One way to look at the effect of the `CMP Ra, Rb` is that the condition register is set as if the value of the difference between the values in `Ra` and `Rb` (Ra − Rb) had been written somewhere.

I chose to have the CPU do a *signed* compare. That means that the CPU assumes that the 2's complement convention for representing negative numbers is used. It considers any number for which the most significant bit is 1 to be negative. Thus the number `8100H` would be considered to be smaller than `7900H` or `79FFH`. Some CPU's also implement an unsigned compare, but the PFW007 does not.

Back to using the `BR` instruction, to specify the condition for jumping the corresponding bits of `Cond` should be set to one and the rest to zero. The jump will occur only if at least one bit which is one in `Cond` is also one in the condition register. For example, if we want to jump only if the last number stored is zero, we would use a `Cond` operand of `1`. If we wanted to jump to address `01A2` only if the value in

register R1 is less than or equal to that in register R3, we would use CMP R1, R3 followed by BR 05, 01A2.

14.6.3 Arithmetic Operations

The ADD instruction adds the contents of two registers and places the result in the first register. The INC and DEC instructions respectively add and subtract one from the contents of a register. The LSHIFT and RSHIFT instructions shift the contents of a register one place to the left and right, respectively. For example, if the register R3 contained the binary number 1001001001001001, then the LSHIFT and RSHIFT instructions would have the effects shown in Table 14−2.

SHIFT EFFECTS

	LSHIFT	RSHIFT
Before	1001001001001001	1001001001001001
After	0010010010010010	0100100100100100

TABLE 14−3. Effects of left and right shifts on the contents of the register for a particular number stored initially in it. For clarity, the contents of the register are shown in binary.

The shift instructions are used for a number of purposes. Perhaps one of the less obvious is multiplication or division by two. Left-shifting a number *n* places is equivalent to multiplying it by 2^n, and right-shifting is equivalent to dividing by 2^n.

The general purpose registers hold 2-byte numbers. The SWAB (for swap byte) instruction can be used to swap the order of the bytes. For example, if register R1 contains 12AB, executing the instruction SWAB R1 changes the contents to AB12.

14.6.4 Logical Operations and HALT

The AND and OR instructions perform a logical AND and OR, respectively, of the contents of two registers, placing the result in the first register. The phrase "a logical AND" of two numbers means that each bit of the result is the logical AND of the corresponding bits of the two numbers. The phrase "a logical OR" means the same thing except that the bits are logically OR'ed. For example, if register R0 contained the number 1001001001001001, and R1 contained 0000000011111111, then the results of AND R0, R1 and OR R0, R1 are shown in Table 14−3.

EFFECTS of AND and OR

		AND R0, R1	OR R0, R1
R0	Before	1001001001001001	1001001001001001
R1		0000000011111111	0000000011111111
R0	After	0000000001001001	1001001011111111
R1		0000000011111111	0000000011111111

TABLE 14−4. Effects of AND'ing and OR'ing the contents of two registers. For clarity, the contents of the registers are shown in binary.

Notice that the contents of the second register are not disturbed. The effect of the NOT instruction is to take the one's complement of the number in the register. That means that every bit that's one is changed to zero, and every zero is changed to a one.

The HALT instruction causes the processor to stop executing instructions. It should be the last instruction of every program.

14.6.5 The Machine Instruction Column

The column giving the machine language version of the instructions is cryptic and needs explanation. The machine code for all instructions is given in hex, and is either two hex digits (one byte) or six hex digits (three bytes) long. This column tells you how to form each hex digit. The left-most (most significant) digit specifies the general type of operation, and the second digit gives further information. The last four digits of six-digit instructions specify an address or a 2-byte number.

In many cases the second hex digit specifies which registers are involved. To do so the registers are labeled by numbers from 0 to 3, corresponding to `R0` through `R3`. For instructions involving two registers, such as the register-to-register move the second byte is formed by multiplying the number of the first register by four and adding it to the number of the second. For example, the instruction with assembly language mnemonic `MOV R2, R1` would have the machine code `19` where $9 = 4 \times 2 + 1$. If the first register had been `R3`, the machine code would have been `1D`.

Instructions involving only one register are a little simpler. For example, the machine code for `LSHIFT R2` would be `8A`. To specify a location in memory, the 16-bit address of the location must be given in hexadecimal, and in the third column of the table this is referred to as *aaaa*. The *a*'s refer to the individual hex digits of the address. For example, the instruction `MOV R1, 0C24` copies the contents of memory address `0C24` to register `R1`, and the machine code is `290C24`. The second hex digit is obtained as directed by the table by adding 1 (from the register number) to 8. You can also load a register with a specific number. This number must be given in hexadecimal, and in the third column it is called *nnnn*. Thus to load `R2` with `C12H` the assembly language instruction would be written `MOV R2, #0C12`, and the machine code would be `220C12`.

14.6.6 Some Simple Program Examples

This instruction set is fictional; as far as I know there is no CPU that uses it. The set is typical, however, of the instruction sets of most CPU's. There are a number of features of real CPU's that aren't implemented in the PFW007, but the basic ideas are there, and you can write useful programs using it. In the next chapter we'll discuss some such programs, but here I just want to give a few examples of how the instructions work.

For example, to add 3 and 5 and put the result at location `100H` of memory, we could use the following program. Here I put the address, the assembly language version, and the machine language version of each instruction on the same line.

5 + 3 = ?

Address	Assembly Language	Machine Language
0000	MOV R0, #3	200003
0003	MOV R1, #5	210005
0006	ADD R0, R1	71
0007	MOV 100, R0	240100
000A	HALT	00

Figure 14−2. A simple program to add 3 and 5 and store the result in memory address `100H`. The Address column gives the memory address where the instruction is stored.

Suppose we want to subtract two numbers. The instruction set doesn't include a subtract instruction so we have to improvise. One way to subtract 3 from 5, for example, would be to load 3 and 5 into registers, change 3 to −3, and add. There is no instruction to change the sign of a number either, but recall that a number can be changed to its negative by taking the 2's complement. There's also no instruction to do that,

but the 2's complement can be formed by inverting each bit and then adding one. We can invert each bit with the `NOT` instruction, and we can add one using the `INC` instruction. Thus the program would be

5 − 3 = ?

Address	Assembly Language	Machine Language
0000	MOV R0, #3	200003
0003	MOV R1, #5	210005
0006	NOT R0	B4
0007	INC R0	80
0008	ADD R0, R1	71
0009	MOV 100, R0	240100
000C	HALT	00

Figure 14−3. Simple program to subtract 3 from 5 and store the result in memory address `100H`.

I've written a program which runs on real computers (the machines in the computer lab) and emulates the PFW007, and which you can use to check these two programs or to try your own hand at writing programs for the PFW007. I'll discuss this program and how to use it in the next chapter.

14.7 Higher Level Languages

With time, assemblers became more helpful. One innovation allowed the programmer to assign a symbolic name to a specific location in memory, and refer to this locations by its symbolic name rather than its address. For example, if you were writing a program to sum a set of numbers entered one-at-a-time from the keyboard, you would want to set aside two areas of memory, one to store the total up until now, and the other to store the number entered from the keyboard. It would be more convenient to refer to these locations as, say, `total` and `nextnum` than as their addresses, say `3D28F104` and `3D28F108`. (I've expressed the addresses in hexadecimal.)

Another innovation was to allow instruction labels. Any instruction in a program could be labeled, and a jump statement could then make use of this label rather than the actual address of the statement. For example, in the program above which adds a sequence of numbers entered from the keyboard, the program needs to know when to stop inputing numbers so that it can print out the result. You might decide to tell it when you were finished entering numbers by entering a zero. The program might then test `nextnum` to see if it is zero, and jump to an instruction labeled `done` if it is.

Even with all these improvements, programming in assembly language is tedious and error-prone. In the early 1950's, the FORTRAN compiler program appeared. This program processes input written in a language with a different vocabulary and grammar than assembly language, called FORTRAN. The idea here was to develop a language which uses instructions more closely related to tasks the programmer would want to carry out than to the instruction set of the CPU. The compiler program reads in the programmer's program written in FORTRAN and translates it into an equivalent program in the machine language of the CPU on which the compiler runs. Thus the person gets to "speak" FORTRAN, whereas the CPU continues to "speak" it's own arcane machine language.

FORTRAN was designed primarily with the scientific and engineering users in mind, and is adept at doing arithmetic and evaluating formulas. In fact the name, FORTRAN, stands for FORmula TRANslator. FORTRAN statements look more like algebra than assembly language, and often one FORTRAN statement translates into several machine language instructions. When programming in FORTRAN, you need not worry about details such as the addresses used to store numbers. For example, the following program stores the integer `3` in a memory location which can be referred to by the name `NUM1`, the integer `8` in a

location which can be referred to by NUM2, and stores the sum of the two in a third location, NUMSUM. The program is closely analogous to the PFW007 program in Fig. 14−2.

5 + 3 = ?

```
NUM1 = 3
NUM2 = 8
NUMSUM = NUM1 + NUM2
```

Figure 14−4. Adding 5 and 3 and storing the result in NSUM using FORTRAN.

This is obviously easier to write and understand than either the assembly or machine language version of essentially the same program.

At least for many applications, the ideas behind FORTRAN turned out to be good ones. FORTRAN is still commonly used, and there are probably more science and engineering programs written in FORTRAN than in any other language. There are at least two reasons for its success. The first is that a FORTRAN program is much closer to the way people think than is assembly language (or even worse machine language). With FORTRAN one has to worry about many fewer details than in the lower-level languages, and can spend more time figuring out how to solve the problem.

Another advantage of FORTRAN which might not be quite so evident is that programs written in FORTRAN are much more portable from one type of computer to another than are programs written in assembler. Different types of computers have CPU's with different instruction sets, so an assembly language program written for one CPU has no chance of working with another. With a higher-level language like FORTRAN, the details of the CPU instruction set are hidden. If you want to add two numbers, the FORTRAN statements needed to do it are the same on an Apple MacIntosh, an IBM PC, and a DEC VAX, even though the CPU's of the three computers are very different. Thus to create a program for each of these computers, you would only need to write one program in FORTRAN. Assuming you have a FORTRAN compiler program (that's a program that translates the program written in FORTRAN into its machine language equivalent) for each machine, you would then run your program written in FORTRAN through each compiler. Doing so would produce three executable programs, each using a different machine language code. When run on the corresponding computer all three codes would accomplish the same task.

Since the advent of FORTRAN a number of other languages have appeared. Some are widely used today, some have all but disappeared. Another early language was COBOL. COBOL was developed for things like accounting and record keeping. The intent was to make a COBOL program read as much like English sentences as possible. While it may be questionable how well COBOL fulfills this requirement, it has been a big success, and it is still in common use. There are lots of other languages. A few of the more common include BASIC, PL/I, APL, Pascal, LISP, SNOBOL, Ada, and C. PL/I, SNOBOL, and APL have lost popularity and are not seen much any more, although they all have their disciples. Pascal also seems to be on the decline, but it's still around.

BASIC, as the name implies, started as a fairly small language intended for writing "quick and dirty" programs. There are a number of rather different versions around, and some of these versions have become quite "non-basic." Because of all these versions, BASIC is very poorly standardized, and a program written in one version may require a lot of work before it can be run on a computer using another version. Although I was at one time pretty fluent in one version of BASIC, I find programs written in other versions difficult to understand, and I would not even consider trying to write a program in these versions without spending quite a bit of time with the language manual.

LISP is a language that became very popular with people trying to develop artificial intelligence (that means computer programs that act as if they can think). I don't know much about the language, other than

I find it very frustrating to study. Perhaps more than any other language, you either love LISP, or you hate it. A member of the latter camp, I'm sure, once said that LISP is an acronym standing for "Lots of Idiotic, Stupid Parentheses."

Ada is a relatively new language that was designed by the U.S. Department of Defense. The jury is still out on Ada. In spite of a requirement that all programs written for sale to the Department of Defense be written in Ada, it seems to be catching on very slowly. I don't know much about Ada, but the most common complaint about it is that it has practically every feature known to the human race, and some of the compromises required to fit all that into one language cause problems. Alan Sherman once quipped that "A camel is a horse that was put together by a committee." The specification for Ada was put together by a committee...

C is a language that was developed by two workers at Bell Laboratories in the early 1970's. The C language has grown in popularity explosively during the past ten years or so. The reason for this popularity is that C offers most of the advantages of other high-level languages while retaining most of the capability of assembly language. Thus, it is reasonable to write in C almost any program that would otherwise have been written in assembler because of the inadequacies of other high-level languages. Because C is so much more convenient than assembly language, almost all such programs are written today in C. The resulting programs are generally a little less efficient than a well written assembly language program, but they usually work better because C relieves the programmer of most of the burdensome details required in assembly language programming. Besides these uses, C is at least competitive with FORTRAN for scientific and engineering computations. Much of the remainder of the course will be devoted to learning and using C.

Although other high-level languages such as FORTRAN are very helpful for writing many types of programs, they do have significant limitations which C largely overcomes. Because statements in these languages are so far removed from the actual instruction set of the CPU, there are some things that are very difficult or clumsy to program in them, but are easy to program in assembly language. For example, in writing a program to control a disk drive, one must be able to write to and read from registers in the disk drive interface circuit. For many computers, doing so requires the ability to transfer data to or from a specific address. FORTRAN, for example, does not contain any statement that allows one to do this, but the operation can be carried out easily using the `MOV` assembly language instruction. Other things are just more clumsy than they need be. For example, one sometimes wants to shift all the bits in a number several places to the left or right. Such an instruction is included in the instruction set of most CPU's, but FORTRAN and many other high-level languages contain no statement to do so directly. Such a shift is equivalent to multiplying or dividing by a power of two, which these languages can do, and one could accomplish the shift this way. A program written using this trick would work, but it's clumsy and obtuse. It may be difficult for someone else (or even the programmer several years later) to figure out how it works. Further, unless the compiler is very well written, it will probably actually carry out the multiplication or division rather than just using the shift instruction, so the program will be slower than it need be. On the other hand, some FORTRAN compilers are indeed very well written.

In this course, I'll be trying to "kill two birds with one stone" by using C as a high-level assembler so that we can explore the operation of a computer at a fundamental level, and by using C as a high-level, "number crunching" language that you can use for computations. Previously, we taught both FORTRAN, and assembly language. Students needed FORTRAN for later classes and assembly language for lab classes in which they designed and built circuits using microprocessors. For both applications C is a good alternative.

14.8 Appendix: The 80386 Architecture

Programming the family of microprocessors to which the 80386 belongs at the assembly language level is complicated because of several details unique to this family of microprocessors. The complications are caused by the self-imposed requirement of compatibility with earlier microprocessors in the same family. Compatibility means that machine language programs that work on one member of the family must

also work on all subsequent members. The 80386 was required to be compatible with the 80286, which had to be compatible with the 8086, which had to be compatible with the 8080. The 8080 was one of the first 8-bit microprocessors, and at the time it was introduced it was a good design. Because of the compatibility requirement, many of the features of the 8080 are still present in the 80386, even though technology has advanced to the point that they no longer make much sense.

One relatively benign example of the problem is the architecture of the general purpose registers of the 80386. The 8080 had four 8-bit registers called `A`, `B`, `C`, and `D`. The next scion of the family, the 8086, had 16-bit registers which were called `AX`, `BX`, `CX`, and `DX`. To maintain compatibility with the 8080, the upper and lower halves of these registers could also be accessed using the names `AH`, `AL`, etc. The 8086 also had four other general purpose registers labeled `SP`, `BP`, `SI`, and `DI`. The next two generations of microprocessors (the 80186 and 80286) were also 16-bit machines, so this scheme held for them as well, but the 80386 is a 32-bit machine. The general purpose, 32-bit registers of the 80386 are called `EAX`, `EBX`, `ECX`, `EDX`, `ESP`, `EBP`, `ESI`, and `EDI`, the same as for the 8086, but with an `E` tacked onto the front end. For compatibility with the 8086, the first 16 bits of each of these registers can also be accessed as if they were separate 16-bit registers by using the old names, `AX`, etc. The first and second bytes of each of the first four of these registers in turn can be accessed as in the 8086 by names like `AH` and `AL`.

This plethora of pseudo registers with seemingly randomly chosen names is more of an annoyance than a real problem. I'm really glad I didn't have to design the instruction decoder in the 80386 that implements this crazy quilt architecture, though! A more troublesome problem is related to what with hindsight seems to me to have been a poor design choice in the 8086. The 8086 was a 16-bit machine, so it could directly handle only 16-bit addresses. With 16 bits you can address only a little more than 64,000 bytes, and even then that was recognized as a potential limitation. Intel gave the 8086 a 20-bit address bus, so it could address up to one megabyte of memory, but there was still a problem because the machine could only do 16-bit arithmetic. The problem was how to specify a 20-bit address using only 16-bit registers and arithmetic.

The solution Intel developed involved adding several more 16-bit registers called segmentation registers. An address is specified by putting a 16-bit number in one of the registers, and then using another 16-bit number in an instruction, like a `MOV` for example. The address that then appears on the 20-bit address bus is made from these two numbers by shifting the number in the segmentation register left 4 places and adding it to the 16-bit number in the instruction. To specify a 20-bit address you have to use two 16-bit numbers, but the process is not quite as onerous as it sounds, because you usually will not have to change the contents of the segmentation register very frequently. For example, suppose you have a block of data starting at some address in memory, say `10A30H`. You could load a segmentation register with the right-shifted starting address, `10A3H`, once and then access say the n^{th} byte using just the number n. For small programs, this scheme works very well, but if your data requires more than 64 K bytes, you will have to modify the contents of a segmentation register to access all of it.

There were other processor families introduced at about the same time as the 8086 which got around the same addressing problem in more satisfactory ways, but Intel's scheme was probably not a bad compromise. In fact, the general idea of being able to move the location of an entire block of data or code around in memory by simply changing the contents of one register has a number of advantages which I won't go into here. The problem is the 64 K limit on the size of a memory segment. The 80386 is a 32-bit machine, so there is no need to keep this 64 K limit, but it's still there because of the requirement of compatibility with the 8086.

14.9 Exercises

1. Write in hexadecimal the machine language version for the PFW007 of the following instructions given in assembly language.

   a. `MOV R3, R1`

   b. `MOV R1, #010A`

   c. `BR 8, #0030`

2. If the registers of the PFW007 hold the following numbers *at the start of each operation*, give the values in all four registers, the program counter, and the condition register after each of the following operations. The values are given in hexadecimal.

   ```
   R0=0020   R1=0f0f   R2=31A4   R3=f0f0   PC=0100   CR=A
   ```

   a. `INC R1`

   b. `MOV R2, #0318`

   c. `ADD R1, R3`

   d. `LSHIFT R0`

   e. `RSHIFT R1`

   f. `SWAB R2`

   g. `AND R1, R3`

   h. `OR R1, R3`

   i. `NOT R0`

   j. `JMP #0160`

   k. `CMP R0, R1`

   l. `CMP R2, R3`

   m. `BR 2, #0030`

   n. `BR 4, #0030`

15. SOME PROGRAMMING TECHNIQUES

In this chapter we'll discuss several examples of programs (well, more nearly program fragments) for the PFW007. The purpose is to show you some common programming ideas, and to give you a better understanding of how the instruction set of a CPU is used to carry out common tasks.

I've written a program that runs on the computers in the lab and emulates the fictional PFW007. This emulator is useful for illustrating how the PFW007 works, and for giving you practice in writing your own programs in machine language. I start the chapter with a discussion of the emulator and how to use it, and then I continue with showing you program examples. You should be able to run these or your own machine language programs on the emulator.

15.1 Using the PFW007 Emulator

The emulator program emulates the fictional PFW007 CPU. The emulator reads a machine language program expressed in hexadecimal, loads it into memory, and executes it. The computer the emulator emulates has only 4096 (= 1000H) bytes of memory. The program prints a line-by-line description of what it did into a file so that you can see if what it did was what you expected.

To use the emulator I suggest you write your program in assembly language on a piece of paper, and then translate it into machine language. After that you have to put it into a file, using your favorite editor. For most of you, I suspect that editor will be the editor included in the Turbo C package, and described in an appendix to this chapter. The emulator expects each byte of your program to be expressed as two hexadecimal digits. You can use either upper or lower case letters for the hex digits A-F. I suggest that you put the program on a floppy disk. The floppy is a little slower than the hard disk, but you can remove it and take it with you.

When reading in a program, the emulator ignores what are called white-space characters. These include space, tab, new-line, and carriage-return characters. The emulator interprets a semicolon, `;`, as indicating the start of a comment, and it ignores all the rest of the characters on that line. I suggest you make use of these features to make your program as readable as possible. I like to put one instruction on each line, and to put the address and assembly language mnemonic for the machine language instruction on the same line as a comment. I do that for my benefit, not the emulator's.

Once you have the program in a file, run the emulator by entering `PFW007` *filename*, where *filename* is the name of the file containing your program. The emulator will read the machine language version of your program from the file, load it into memory, and start executing it. If the emulator encounters any errors either while loading or running your program it prints out a short error message and quits. While running, the emulator prints on the monitor the address of each instruction as it is being executed. These numbers change too fast to read, but they show you that the emulator is busy emulating. If you want to stop the emulator before it halts on its own, press **^C** (remember that's the **Ctrl** and **C** keys pressed simultaneously). When the emulator is finished, it writes out on the monitor the contents of the four general purpose registers.

To let you keep track of exactly what it did, the emulator keeps a line-by-line diary of the instructions it executes and the results in a file. The name of this file is the same as the name of your program file, except the extension is `.RPT`. Thus if you named the file containing your program `MYNOBEL.GO7`, then the report will be placed in a file named `MYNOBEL.RPT`.

As an example of how it all works, Fig. 15−1 contains the simple program I wrote in Chapter. 14 to subtract 3 from 5. I placed the program in a file named `P5MIN3.GO7`. The first line is entirely a comment because the only stuff on the line comes after a semicolon. In the remaining lines the machine language version of each instruction appears at the extreme left end of the line, and the rest is ignored by the

```
; Program to subtract 3 from 5

200003    ;0000:  mov r0, #3
210005    ;0003:  mov r1, #5
B4        ;0006:  not r0       ;Negate r0
80        ;0007:  inc r0
74        ;0008:  add r1, r0
00        ;0009:  halt
```

Figure 15−1. A simple program to subtract 3 from 5 on the PFW007.

emulator because it's either white space or stuff after a semicolon. The comment on each line gives the address of the instruction, followed by the assembly language mnemonic for it, and possibly followed by a comment clarifying what the instruction is supposed to do. This stuff is for my convenience only, the emulator pays no attention to it. The machine language instructions are given in hexadecimal. If you like you can end each hex number with an `h` or `H`, but it is not necessary. One common mistake I make when changing my program is to change the assembly language instruction but not the machine language version of it. It can be very frustrating. I think I've fixed a bug, and the program still behaves as it did before the "fix."

When, at the DOS prompt, I type `PFW007 P5MIN3.GO7` the emulator loads my machine language program into its memory, starting at address `0000`. It then starts executing the program, starting with the instruction at address `0000`. When finished, the emulator prints out the following on the monitor screen.

```
R0=FFFD   R1=0002   R2=0000   R3=0000
```

The value of each or the four general purpose registers is printed out in hexadecimal. Note that in this particular program register `R1` should contain the result, and the number in it is, in fact, correct. A more detailed report of what the emulator did is contained in a file named `P5MIN3.RPT`. The contents of this file are shown in Fig. 15−2. All numbers are expressed in hexadecimal.

```
0000:  MOV R0, #0003:  200003  ;  R0=0003  CR=A
0003:  MOV R1, #0005:  210005  ;  R1=0005  CR=A
0006:  NOT R0       :  B4      ;  R0:  0003 -> FFFC  CR=6
0007:  INC R0       :  80      ;  R0:  FFFC -> FFFD  CR=6
0008:  ADD R1, R0   :  74      ;  R0=FFFD  R1:  0005 -> 0002  CR=A
0009:  HALT
```

Figure 15−2. Listing showing the report generated by running the program shown in Fig. 15−1.

Each line contains four items. The first is the address in hex of the instruction being executed. The instruction itself expressed in assembly mnemonics is the second and the same instruction expressed in machine language as read from the input file is the third. Finally, after the semicolon the result of executing the instruction is shown. The symbol `CR` denotes the condition register. For example, the first instruction set the contents of register `R0` to `3`, and as a result the condition register was set to `A`. The third instruction at address `0006` caused the contents of register `R0` to be changed from `0003` to `FFFC`, setting the condition register to `6`.

If the emulator encounters an illegal instruction, it prints out a brief error message and stops. You can get a better idea of what happened by looking at the report file.

The emulator has several special features, which are features of the emulator itself rather than of the instruction set of the PFW007. One involves input to and output from the emulator. If you move a number

from a register to the special memory address `F000` the low byte of the number will be written into a file with the same name as your program file, but with extension `.OUT`. This file will be created as needed. If you do the opposite, and move a number from `F000` to a register the emulator will read one byte from a file and return it as the lower byte of the number moved. The name of the file for input is the same as that of the program file, but with extension `.IN`. It is an error to try to move a number from this special address if this input file does not exist. If the emulator comes to the end of the file, and you try to read another byte from it, the emulator returns a negative number. Thus, you can check whether or not you have reached the end of the file by checking the sign of the number returned.

By default, the loader loads the bytes of the program file into memory at sequentially-increasing addresses, starting at `0000`. The first byte is put in `0000`, the second in `0001`, etc. You can change where subsequent bytes in the program file are placed in memory using the `.ORG` directive. The directive has the form

```
.ORG aaaa
```

The period should be the first character on the line, and the four `a`'s denote the address (in hexadecimal) where the first byte of the next line in the file is to be placed. From this point onwards, the bytes in the file will be stored in sequentially-increasing addresses in memory, just as before, but with the new origin. (That's where the `ORG` mnemonic came from—ORiGin.) You can put an `.ORG` directive anywhere in a program file you like, and you can use as many as you like. For example, if the program file contained

```
123456
78
9A
.ORG 20
BCDEF0
```

the loader would place `123456789A` in addresses `0000` through `0009`, and `BCDEF0` in the addresses `0020` through `0025`. If the period were not the first character on the line of the `ORG` directive the emulator would not interpret it as a directive, and would declare the line to be in error.

A related detail is that of how the fictional CPU stores numbers in memory. The registers in the PFW007 are 16 bits or 2 bytes long. When the CPU carries out a `MOV 0100, R1` instruction, for example, how should it store the the two bytes that will be moved to memory? It could store the most significant byte at `0100` and the least at `0101`, or vice versa. Either way would be fine as long as the CPU was consistent and retrieved the data according to the same convention. The same question arises with real CPU's, only with 32-bit machines it's even more troublesome because there are four bytes involved. Some manufacturers choose one convention, some another. The two conventions have names taken, by the way, from *Gulliver's Travels*.[1] Machines which store the least significant byte at the lower address are called *little-endian*, and those which store the most significant byte at the lower address are called *big-endian*. As we will discover in the next chapter, Intel microprocessors are little-endian. In the case of the PFW007, I have chosen to store the most significant byte at the lower memory address and the least at the next higher address, so it is big-endian. I chose this convention because it is consistent with the way the bytes appear in the program file.

Finally, I'd like to discuss a mistake that all programmers occasionally make—creating an endless loop. Consider, for example, the following simple program. shown only in the assembly language. The

---

1. In case you are not familiar with the book, one of the places Gulliver visited was a land in which a civil war was raging. The war involved a dispute over whether one should eat a soft-boiled egg by breaking open the big end or the little end. I guess this war made about as much sense as much of the discussion about which byte ordering was better.

program has a simple but potentially disastrous mistake.

```
0000:  MOV R1, #0006
0003:  DEC R1
0004:  BR 3, #000A
0007:  JMP #0003
000A:  HALT
```

Figure 15−3. Simple program with an error that results in an endless loop.

The intent of the programmer was that the computer would repeat or loop through the instruction addresses `0003` through `0007`, decrementing `R1` each time, until the contents of `R1` become zero. At that time the `BR` instruction is supposed to cause a jump to the `HALT` instruction at the end. The error is in the *Cond* argument to the `BR` instruction. It should have been `1`, but instead it's `3`. As written, the program will jump to the last instruction only if the contents of `R1` are at the same time both zero and non-zero. That will never happen, so the computer will continue looping forever, and you'll never get back to DOS.

Meanwhile, the emulator will be busily writing a line into the report file each time an instruction is executed. If the process were to be allowed to continue, there's the alarming possibility of completely filling the disk with the report file. Because of this possibility, the emulator has a "safety" feature which limits the number of lines in the report file to 1000. There's also a limit to the number of bytes you can write to the `.OUT` file. If you accidentally create an endless loop, it's easy to get out of it. Just press `^C` (that's **Ctrl** and **C** pressed at the same time). That should get you back to DOS where you can look at the report file to figure out what happened.

A final feature of the emulator is that the loader initializes the contents of all memory cells and all but one of the registers to zero before loading in your program. The one register not initialized to zero is the stack pointer register, `SP`, which I will discuss later in section 15−6. Initializing memory to zero has the side-effect that if you mistakenly jump to someplace in memory where there are no instructions the emulator will simply stop because it will see `00`, which it interprets as `HALT`.

15.2 Shifting a Number n Bits and DO Loops

The instruction set of the PFW007 includes instructions to shift the contents of a register either left or right by one bit, but there is no instruction to shift by more than one bit. In this section, we write a program to left-shift a number by a specified number of bits. Call the number to be shifted `x`, and the number of bits to shift it `n`. One way to shift `x` by `n` bits would be simply to put *n* `LSHIFT` instructions into the program. With that approach, however, it is messy to change the number of bits to shift because instructions have to be added or deleted from the program, and doing so changes instruction addresses. Instead, I'll write a program in which both the number and the number of bits to shift are loaded into memory at a convenient location just above the program. Then the number of bits to shift can be easily changed by just changing one 2-byte number in the program file.

We need a strategy—a plan of action. Here's what I came up with. I'll write the program so that it keeps re-executing an `LSHIFT` instruction until it has been executed the required number of times. To determine when the required number of shifts have been performed I'll increment (add `1` to) a register just after each shift, and then I will compare the number in the register to the required number of shifts. If the number is less than the required number, I'll branch back to the `LSHIFT` instruction so that the whole process will be repeated again. When the number of shifts is equal to the required number I'll exit the loop by simply not branching. If I had more to do than just the shifting, the program would then continue, but in this case once the shifting is finished I'm done, so the next instruction after the branch will be a `HALT`.

That's the plan, now I have to put together machine language instructions that will implement it. This is starting to get a little bit complicated, so it's probably a good idea to write the plan down in a form

that's easy to work with. (If the program were much more complicated than this, it certainly would be a good idea!) I usually use a kind of outline. The idea is to get the general plan down, without worrying about the details, and then to go back an fill in the details later. Here's my first shot at an outline.

Lshift `x`
Increment counter
Compare counter with `n`
Branch to start if counter < `n`
Quit

There are a lot of details left out here! For example, the value of `x` has to be copied to a register in order to shift it, and the value of `n` has to be copied to another register to do the compare. We have to assign registers for this purpose, and we have to assign a third register for the counter. Another detail is that we have to make sure the counter register starts off with the correct number in it; otherwise the count will be incorrect. I decide to use `r0` for `x`, `r1` for `n`, and `r2` for the counter. Thinking a little (this is the hard part) I see that the counter should have zero in it initially in order for the count to be correct when the compare execution is executed. Because of the way the emulator works, the registers are initialized automatically with zeroes, so I don't have to do so explicitly, but I choose to do it anyway, just to make the program clear. I'll go back and redo the outline, adding these details to it, so I have fewer details to worry about when we are generating the machine language. Fig. 15−4a shows the revised outline

a)

Put `x` in `r0`
Put `n` in `r1`
Use `r2` for the counter
Load `r2` with `0`
again:
Lshift `r0`
Increment `r2`
Compare `r2` with `r1`
Branch to `again` if `r2` < `r1`
Quit

b)

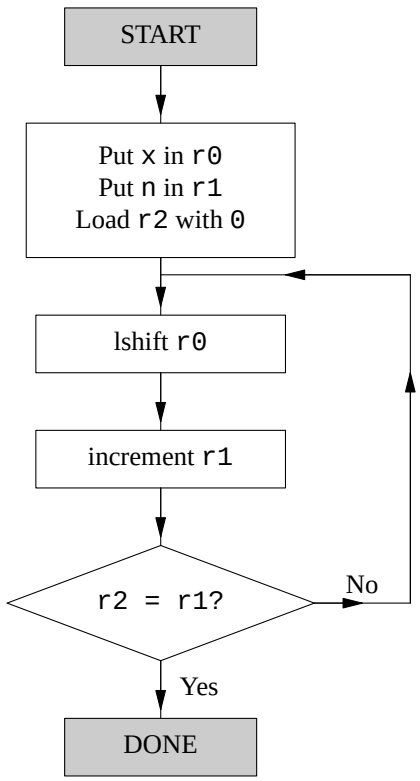

Figure 15−4. Outline (a), and flow chart (b) for the program to left shift a number `x` by `n` bits.

Another common technique is to make a *flow chart*. A flow chart attempts to convey the same

information as an outline, but in a more graphical format. For example, one might make the chart shown in Fig. 15−4b. The lines with arrows show the flow of the program, and the rectangular boxes show actions of the program. Diamond shaped boxes are used to indicate decisions like whether or not the shift counter is still less than the desired number of shifts. Typically there are two lines coming out of the diamond, one for each possible outcome of the test.

Choose whichever technique seems the most useful to you, or develop your own. Just remember why you are writing the outline or flow chart; it's supposed to help you deal with all the bothersome details of getting the program actually written. If the program you are trying to develop is at all complicated, you should plan on spending most of your time in this stage of development. The hard part of writing a program is usually dreaming up a strategy that will work. Once you have a good plan, it is not difficult to translate it into a workable program. I often have to make several outlines before I arrive at one which will work. These notes may be a little misleading in that respect—I only show you the successes. Don't expect to produce a gem the first time, but do expect to produce a gem eventually.

The program I wrote according to this strategy is shown in Fig. 15−5.

```
; Program to left-shift a number stored in memory address 0020
; a specified number of times.  The number of shifts to be performed
; is stored in memory location 0022.
280020          ;0000:  mov r0, x       ; Store x in r0
290022          ;0003:  mov r1, n       ; Store n in r1
220000          ;0006:  mov r2, #0000   ; Initialize counter to 0
        ;again:
88              ;0009:  lshift r0       ; Left shift number in r0 once
82              ;000a:  inc r2          ; Increment shift counter
C6              ;000b:  cmp r1, r2      ; Enough shifts?
580009          ;000c:  br a, again     ; Branch if not
        ;done:
00              ;000d:  halt            ; Finished!
.ORG 0020
        ;x:
00a5            ;0020:                  ; The number to be shifted
        ;n:
0004            ;0022:                  ; The number of times to shift
```

Figure 15−5. Example program which left-shifts a number left by a specified number of bits. On exit, the result is left in `R0`.

Given the outline or flow chart, this program should be pretty much self-explanatory. I had to choose memory addresses for `x` and `n`, and I had to choose values for these variables. The addresses really don't matter, just as long as they are not within the address range containing the program. I chose `0020` for `x`, and `0022` for `n`. To make it easier to check on the program, I chose `n = 4` (why does this make checking easier?), and for no good reason I chose `x = 00A5`. Notice that in the assembly language version, I've not specified addresses explicitly, but rather given each a descriptive name. The emulator doesn't care what I write there because it ignores it completely. I find the names more helpful than the explicit addresses, and the assembly language stuff is for me, not the emulator. In the machine language version I can't get away with these shenanigans, however, and there I must use the addresses explicitly.

The program prints out the result shown in Fig. 15−6a, and creates the report file shown in Fig. 15−6b. The program appears to have worked correctly. It was to left-shift `A5` four bits. Four bits corresponds to one hex digit, so the results should be shifted by one digit. When finished, `R0` contained `0A50`, and and the counter, `R2`, contained `0004`, as expected.

This program could be written a number of different ways. Fig. 15−7 shows another program which performs exactly the same task, but does not use the `CMP` instruction. This program does not maintain a

a)

```
R0=0A50   R1=0004   R2=0004   R3=0000
```

b)

```
0000:  MOV R0, 0020 :  280020  ; R0: 0000->00A5, CR=A
0003:  MOV R1, 0022 :  290022  ; R1: 0000->0004, CR=A
0006:  MOV R2, #0000:  220000  ; R2: 0000->0000, CR=1
0009:  LSHIFT  R0   :  88      ; R0: 00A5->014A, CR=A
000A:  INC R2       :  82      ; R2: 0000->0001, CR=A
000B:  CMP R1, R2   :  C6      ; (R1=0004) > (R2=0001), CR=A
000C:  BR  08, 0009 :  580009  ; CR=A, branch taken
0009:  LSHIFT  R0   :  88      ; R0: 014A->0294, CR=A
000A:  INC R2       :  82      ; R2: 0001->0002, CR=A
000B:  CMP R1, R2   :  C6      ; (R1=0004) > (R2=0002), CR=A
000C:  BR  08, 0009 :  580009  ; CR=A, branch taken
0009:  LSHIFT  R0   :  88      ; R0: 0294->0528, CR=A
000A:  INC R2       :  82      ; R2: 0002->0003, CR=A
000B:  CMP R1, R2   :  C6      ; (R1=0004) > (R2=0003), CR=A
000C:  BR  08, 0009 :  580009  ; CR=A, branch taken
0009:  LSHIFT  R0   :  88      ; R0: 0528->0A50, CR=A
000A:  INC R2       :  82      ; R2: 0003->0004, CR=A
000B:  CMP R1, R2   :  C6      ; (R1=0004) = (R2=0004), CR=1
000C:  BR  08, 0009 :  580009  ; CR=1, branch not taken
000F:  HALT
```

Figure 15−6. Result of running the program in Fig. 15−5.

separate shift counter, but instead decrements the value stored in `R1` each time through the loop. The `BR` instruction just before the end of the program branches back to the start of the loop as long as the value in `R1` after the decrement is greater than zero. This version is a little more indirect that the previous one, but it has the advantage that it has one fewer instruction inside the loop.

The looping structure in both versions of this program is very common in all types of programs. We even encountered a loop in Chap. 6 when writing a macro to help analyze Herbie's data. Such structures are often called *DO loops*. I think the name comes from FORTRAN; the FORTRAN instruction to set up such a loop is `DO`. You will encounter these structures frequently in the remaining chapters of these notes, and in programming of all types.

15.3 Writing to a File and Character Strings

In this example I want to program the PFW007 to write a name into the output file. (I chose "Joe" instead of my name because "Joe" makes the report file shorter.) Through this example, I'll be able to show you how higher-level languages handle textual data, and we'll splash around a bit in a small mud puddle resulting from the fact that characters are stored as single bytes, but the PFW007 moves two bytes at a time. The program also illustrates the use of indirect `MOV` instructions.

We have several problems to solve and decisions to make in order to write the program. Here are the solutions I chose and decisions I made. Others are certainly possible. One problem is how to represent the characters in the name. The name is to be loaded into some convenient location in memory when the program is loaded, and then written to a file when the program is executed. The most conventional way of representing letters is to use the ASCII codes for the letters. We discussed the ASCII code back in Chapter 1. For your convenience, I've included a table giving the ASCII equivalents for all the letters and some other characters in hexadecimal in Table 15−1. I chose to use the name "Joe" for demonstrating the program. Then the ASCII representation of the name consists of three bytes, `4AH`, `6FH`, and `65H`.

```
; Another program to left-shift a number (x) a variable
; number (n) of places.

280020      ;0000:  mov r0, 0020    ; Store x in r0
290022      ;0003:  mov r1, 0022    ; Store n in r1
    ;again:
88          ;0006:  lshift r0       ; Left shift number in r0
85          ;0007:  dec r1          ; Decrement n
580006      ;0008:  br 8, again     ; Branch if greater than zero
00          ;000b:  halt            ; Finished!
.ORG 0020
    ;x
00a5        ;0020:                  ; The number to be shifted
    ;n
0004        ;0022:                  ; The number of times to shift
```

Figure 15−7. A program which performs the same function as that in Fig. 15−5.

Here's how the program I dreamed up will work. I'll load the address of the start of the name into a register. To make the discussion easier, I'll choose `r0` for this register right now. Then I'll use the indirect move instruction of the PFW007 to move the number stored at the address contained in `r0` to the I/O address, `F000`. Then I'll increment the address in `r0` and branch back to the indirect move instruction. Here, I run into a little problem, though. How does the program know when it has copied all the letters of the name to the file? The problem of detecting the end of a character string is a common one, and there are a number of good solutions to it. One common solution is to store an extra number with the character string which gives the number of characters in the string. With this convention, the first location in the memory block containing the name would contain the number `3` (there are three characters in "Joe"), followed by the ASCII codes for "Joe". If we choose to use two bytes to hold the number of characters, then a total of five bytes would be required, `00H`, `03H`, `4AH`, `6FH`, `65H`. To implement our program with this convention, we would load the first two bytes into a register, and decrement the register each time a byte is transferred. The contents of the register would be checked each time, and the program would branch backwards only if the contents were greater than zero.

Another popular convention is to agree on an "end-of-string" character which is used for nothing else, and to end all strings with this character. Zero is used almost universally for this purpose. Thus, "Joe" would require four bytes with this convention, `4AH`, `6FH`, `65H`, `00H`. To implement our program using this convention, we would test each byte before transferring it to the file. If the byte is non-zero it should be transferred and another byte fetched. If zero, execution of the program should cease.

I choose to use the second convention, primarily because it is the one most commonly used with programs written in the C programming language. Fig. 15−8 shows an outline of the program.

Thinking about the details of this plan, we fall into the small mud puddle I mentioned earlier. The `MOV` instruction labeled `again` in the outline moves two bytes into `R2`, placing the byte at the address contained in `R0` into the high or most significant position in `R2`, and the byte at the next higher address into the low or least significant position. If we were to write the contents of `R2` to the I/O address only the low byte would be written to the file. This is the ASCII code not for the first letter, but for the second. One solution to the problem would be to load `r0` initially with the address of the byte just before the start of the name. That way, the correct byte would be written to the file each time. This solution works pretty well, but it is a bit tricky and it would be difficult for someone else to figure out how it works.

I chose a more direct solution. My program initializes `r0` with the correct start address, but it uses a `SWAB` instruction to swap the low and high bytes so that the desired byte is in the correct place. Just to make sure, I decided to explicitly zero out the high byte after the swap. That's not really necessary since the `MOV` to the I/O address moves only the low byte. I zeroed the unwanted high byte by `AND`'ing `r0` with a number for which the low 8 bits are all `1`'s, and the high 8 bits are all `0`'s. (In hex that number turns out

ASCII CODES IN HEXADECIMAL

Char.	Code	Char.	Code	Char.	Code	Char.	Code
^@	00	Space	20	@	40	'	60
^A	01	!	21	A	41	a	61
^B	02	"	22	B	42	b	62
^C	03	#	23	C	43	c	63
^D	04	$	24	D	44	d	64
^E	05	%	25	E	45	e	65
^F	06	&	26	F	46	f	66
^G (Bell)	07	'	27	G	47	g	67
^H (BS)	08	(	28	H	48	h	68
^I (Tab)	09	)	29	I	49	i	69
^J (NL)	0A	*	2A	J	4A	j	6A
^K	0B	+	2B	K	4B	k	6B
^L	0C	,	2C	L	4C	l	6C
^M (CR)	0D	-	2D	M	4D	m	6D
^N	0E	.	2E	N	4E	n	6E
^O	0F	/	2F	O	4F	o	6F
^P	10	0	30	P	50	p	70
^Q	11	1	31	Q	51	q	71
^R	12	2	32	R	52	r	72
^S	13	3	33	S	53	s	73
^T	14	4	34	T	54	t	74
^U	15	5	35	U	55	u	75
^V	16	6	36	V	56	v	76
^W	17	7	37	W	57	w	77
^X	18	8	38	X	58	x	78
^Y	19	9	39	Y	59	y	79
^Z	1A	:	3A	Z	5A	z	7A
^[(Esc)	1B	;	3B	[	5B	{	7B
^\	1C	<	3C	\	5C	|	7C
^]	1D	=	3D	]	5D	}	7D
^^	1E	>	3E	^	5E	~	7E
^_	1F	?	3F	_	5F		

TABLE 15−1. Table of ASCII codes expressed in hexadecimal. The following abbreviations are used: BS = backspace, NL = new line, CR = carriage return, Esc = Escape.

to be `00FF`.) Then the low 8 bits in `r0` are AND'ed with `1`'s, and the result is whatever value the bit had before the AND'ing. The high 8 bits are AND'ed with `0`'s, and the result is `0`, as desired. Fig. 15−9 shows the program with these modifications. I chose to put the name in memory starting at `0020`, and put the masking number `00FF` in `r1`.

I named the file containing the program `PROG9.GO7`. Running it created a file named `PROG9.OUT` which contained a single line,

```
Joe
```

The report file is returned in `PROG9.RPT` shown in Fig. 15−10. As you can see, the program worked as

Load `r0` with address of the start of the string
again:
Move byte at this address to `r2`
Branch to end if this byte is zero
Move the byte to I/O address, `F000`
Increment address in `r0`
Jump to `again`
end:
Halt

Figure 15−8. Outline of the proposed program to copy a name stored in memory to a file.

intended.

```
; Program to copy byte-by-byte ascii chars
; from memory to output file.

200020    ;0000:  mov r0, name     ; Pointer to name in r0
2100ff    ;0003:  mov r1, #00ff    ; For masking off high byte
    ;next:
48        ;0006:  mov r2, @(r0)    ; Get a char from memory
b2        ;0007:  swab r2          ; Interchange bytes in r2
99        ;0008:  and r2, r1       ; Mask off high byte
510013    ;0009:  br 1, done       ; If char is 0, finish
26f000    ;000c:  mov f000, r2     ; Write it to file
80        ;000f:  inc r0           ; Increment pointer to next char
2C0006    ;0010:  jmp next         ; End of loop
    ;done:
00        ;0013:  halt
.ORG 0020
    ;name:
4A        ;0020:  J
6F        ;0021:  o
65        ;0022:  e
00        ;0023:  End of String    ; End is marked with a zero
```

Figure 15−9. Program listing for the PFW007 which copies the characters starting in memory at `0020` to the output file. Copying terminates when the program encounters a zero byte.

This example illustrates several common features of computer programs. First, it shows how textual data is stored and manipulated in a computer. The individual characters are translated to their ASCII equivalents, and stored sequentially in a block in memory, with the first character at the start of the block. The program illustrates a common technique for marking the end of a character string. This is the "standard" method used with C programs. The program also provides an example of the use of a pointer. The value stored in register `R0` is not itself a datum of direct interest, but rather it's the address in memory of such a datum. In C such a quantity is called a *pointer*. It points to an item of interest, the next character in the string in this case. Using a pointer, we were able to step along from one character to the next in the string by simply incrementing the pointer. An alternative would have been to keep in a separate register an index labeling the current character. To get the address of the corresponding character, we would add the index to `name`, the starting address for the string. Stepping through the character string would then involve incrementing this index rather than the pointer. The method chosen in the program is obviously simpler and faster for this application but it would not have worked so well if we had wanted to access the characters in a non-sequential order.

```
0000:  MOV R0, #0020:  200020  ;  R0=0020, CR=A
0003:  MOV R1, #00FF:  2100FF  ;  R1=00FF, CR=A
0006:  MOV R2, @(R0):  48      ;  R2=4A6F, CR=000A
0007:  SWAB R2       :  B2      ;  R2:  4A6F -> , CR=A
0008:  AND R2, R1    :  99      ;  R1=00FF  R2:  6F4A -> 004A, CR=A
0009:  BR  01, 0013 :  510013  ;  CR=A, branch not taken
000C:  MOV F000, R2 :  26F000  ;  @(F000)=004A, CR=000A
000F:  INC R0        :  80      ;  R0:  0020 -> 0021, CR=A
0010:  JMP 0006      :  2C0006  ;  PC:  0010 -> 0006
0006:  MOV R2, @(R0):  48      ;  R2=6F65, CR=000A
0007:  SWAB R2       :  B2      ;  R2:  6F65 -> , CR=A
0008:  AND R2, R1    :  99      ;  R1=00FF  R2:  656F -> 006F, CR=A
0009:  BR  01, 0013 :  510013  ;  CR=A, branch not taken
000C:  MOV F000, R2 :  26F000  ;  @(F000)=006F, CR=000A
000F:  INC R0        :  80      ;  R0:  0021 -> 0022, CR=A
0010:  JMP 0006      :  2C0006  ;  PC:  0010 -> 0006
0006:  MOV R2, @(R0):  48      ;  R2=6500, CR=000A
0007:  SWAB R2       :  B2      ;  R2:  6500 -> 0065, CR=A
0008:  AND R2, R1    :  99      ;  R1=00FF  R2:  0065 -> 0065, CR=A
0009:  BR  01, 0013 :  510013  ;  CR=A, branch not taken
000C:  MOV F000, R2 :  26F000  ;  @(F000)=0065, CR=000A
000F:  INC R0        :  80      ;  R0:  0022 -> 0023, CR=A
0010:  JMP 0006      :  2C0006  ;  PC:  0010 -> 0006
0006:  MOV R2, @(R0):  48      ;  R2=0000, CR=0001
0007:  SWAB R2       :  B2      ;  R2:  0000 -> 0000, CR=1
0008:  AND R2, R1    :  99      ;  R1=00FF  R2:  0000 -> 0000, CR=1
0009:  BR  01, 0013 :  510013  ;  CR=1, branch taken
0013:  HALT
```

Figure 15−10. Report file for program in Fig. 15−9.

Finally, the program provides an example of the use of an array. The characters making up the string "Joe" are stored in a block of memory between `0020` and `0023`. Such a block is called an *array*. In our program, "Joe" was stored in an array of characters, each one byte long. In C, arrays can be given any name you like, and the value associated with the name is a pointer to the start of the array. You can access an element of the array either by using a separate pointer, or by adding an integer to the name of the array. For example, if `name` is the name I give the character array containing "Joe", then I could access the 'o' either by using a pointer set with value `0021`, or by adding one to `name`.

15.4 A Program to Multiply Two Numbers

In this section we consider a somewhat more complicated example, a program to multiply two unsigned integers. Before jumping into writing the program, we first have to figure out how we will do the multiplication—we need to figure out an *algorithm*. We looked at the multiplication of binary numbers in EE 121, and this would be a good time to review that material. Briefly, the idea is the same as that for multiplying two numbers in decimal by hand. First you write one number above the other. The numbers have special names which I can never remember, so I'll call the number you put on top the "multiplyee", and the one on the bottom the "multiplior." (I'll bet you didn't know I speak legalese.) Then you multiply the least significant digit of the multiplior times the multiplyee, and write the result down somewhere. In binary this is particularly easy—the result is either zero, or the multiplyee. After that, you multiply the second digit of the multiplior times the multiplyee and write that result down below the first result and shifted one place to the left. You continue doing that until you run out of digits in the multiplior, and then you add up all the results, keeping the shifts. That's the answer.

The program I'll write will use the same idea. I'll load the multiplyee into `R0`, and the multiplior into `R1`. I'll use `R2` to hold the accumulated sum. (Instead of keeping track of all the individual one-digit multiplications and adding at the end I'll keep a running sum, adding as I go.) First I'll initialize `R2` to

zero. Then I'll check the least-significant bit in the multiplior and add the contents of R0 to R2 if it's one. If the bit is zero, I won't do anything to R2. After that, I'll left shift R0 and right shift R1 one place. Again, if the least-significant bit in R1 is now one I'll add the contents of R0 to R2, otherwise I'll leave R2 alone. I'll continue this process until I've shifted all the ones in R1 out of the register and R1 contains zero. When I get to that point I'm done, and the answer is in R2.

Fig. 15−11 shows the program to do all this. I'll not give the program outline or flow chart, but you should believe that I went through several outlines before I came up with a reasonable one. I chose to put the multipliee in memory address 0040, and the multiplior in 0042. To check the program, I used 00A3 (163 in decimal) for the multiplyee, and 002C (44 in decimal) for the multiplior.

```
; Program to multiply two unsigned integers

280040    ;0000:  mov r0, mpee         ; r0 will hold the multipliee
290042    ;0003:  mov r1, mpor         ; r1 will hold the multiplior
220000    ;0006:  mov r2, #0000        ; r2 will accumulate the sum
    ;again:                           ; Start of the loop
230001    ;0009:  mov r3, #0001        ; Mask for checking L.S.B.
9d        ;000c:  and r3, r1           ; Is L.S.B. of r1 zero?
510011    ;000d:  br 1, bit_is_zero    ; Branch if yes
78        ;0010:  add r2, r0           ; Otherwise update sum
    ;bit_is_zero:
88        ;0011:  lshift r0
8d        ;0012:  rshift r1
                                      ; End of loop,
520009    ;0013:  br 2, again          ; do again if r1 not 0
    ;done:
00        ;0016:  halt
.ORG 0040
    ;mpee:
00a3      ;0040:  multipliee
    ;mpor:
002c      ;0042:  multiplior
```

Figure 15−11. Program to multiply two unsigned integers in memory addresses 0040 and 0042.

The program runs and prints out the following.

```
R0=28C0   R1=0000   R2=1C04   R3=0001
```

The value in R2 should be the result of multiplying A3 times 2C. Multiplying the decimal equivalents gives 7172 in decimal. The value in R2 is in decimal 7172, so it works!

There are several areas where improvement is needed for a practical program. The first improvement is an efficiency improvement. The smaller the multiplior, the fewer times the program must go through the loop. The program should, therefore, test the sizes of the two numbers to be multiplied and interchange them if needed to put the smaller one on the bottom (the multiplior). A more serious problem is that there is no provision made for handling negative numbers. The algorithm will give wrong results if either of the numbers is negative. We could fix this by testing each number to see if it's negative, and 2's complementing it if so. We would have to remember how many of the numbers were negative, and change the sign of the final result if one but not two were negative.

Another serious problem is the possibility of *overflow*. If the product of the two numbers is larger than FFFF in hex, or 65535 in decimal, the product will require more than 16 bits. This program will simply shift these extra bits off the left end of the register and give us the wrong answer without any indication that something is wrong. This is a more difficult problem to fix. Many CPU's reserve a bit in the condition

register for overflows. If you shift a bit off the left end of a register, or add two numbers with a sum too large to fit in a standard word, the overflow bit is set. Using an overflow bit doesn't fix the problem, but at least it lets you find out that it occurred. That solution isn't available with the PFW007. Another procedure would be to rewrite the program so that the result is placed in two registers. Then overflow could not occur. That would be possible, but the program would get pretty complicated. For these notes, I'll just be satisfied with the program as it is.

15.5 Subroutines

Often when writing a program you need a smaller subprogram which accomplishes some set task, and which may be used several times in your program.. For example, you very well might need to multiply numbers at several places in a program. One way to handle such a situation would be to put in a block of code similar to that in the last section each time you needed to do a multiplication. That could make your program quite a bit longer, especially if you included some of the improvements suggested at the end of the section, and it would be tedious to do because you would have to go through each block and change the addresses in it accordingly. Also it would make your job of programming, and of debugging the program, more difficult because the blocks of multiply code interrupt your train of thought. It's more convenient to just think of the block as a "black box" that multiplies two numbers without having to remember how it works.

For these reasons programmers often make use of *subroutines*. Continuing with the multiply example, we might modify the multiply program we wrote in the last section so that it could be used as a subroutine. Once that is done, every time we want to multiply two numbers, we would only need to put them in the proper registers (or wherever the subroutine expects to find them), and jump to the start of the subroutine. Things should be designed so that the subroutine jumps back to the instruction in the main program after the one that called it when it finishes. The use of subroutines is so common that practically all CPU's include an instruction to jump to a subroutine, and another instruction normally placed at the end of the subroutine which causes the CPU to jump back to the correct place in the main program. The PFW007 has such instructions, and the use of these instructions along with the use of subroutines is the subject of this section. The instructions are `JSR` and `RTS`.

In this section we will convert the multiply program into a subroutine, and then use the subroutine in a simple main program to multiply two numbers. The example will accomplish the same task as the "subroutine-less" program in 15−11, but it will bundle the complexity associated with the actual multiplication into a separate package (the subroutine) which we need not consider in any detail to write or understand main program. This is one of the advantages of using subroutines. The other advantage of using subroutines is not illustrated by this example because the main program is unrealistically simple. Suppose instead the main program required numbers to be multiplied at several places. With the use of the multiply subroutine, each multiply would require only two `MOV` instructions to put the two numbers to be multiplied (the multiplicands) into the proper registers, a `JSR` instruction to execute the code in the subroutine, and maybe another `MOV` instruction to put the answer wherever we want it. Without using subroutines, each multiply would require copying the code in Fig. 15−11 into our program, making the appropriate changes to addresses. Doing that is considerably longer and more tedious than using the subroutine.

The modified program is shown in Fig. 15−12. The subroutine starts at memory location `0100`, and the main program (the one which calls the subroutine) is located before it, starting at `0000`. The main program is very simple, it simply puts the two numbers to be multiplied where the subroutine expects to find them (I chose to use registers `R0` and `R1`), and then executes a `JSR` (`J`ump to `S`ub`R`outine) instruction to jump to the start of the subroutine. When the subroutine is finished, control is returned to the instruction immediately following the `JSR` instruction, a `HALT` in this case. In a more realistic case, the main program would probably continue, making use of the result returned in register `R2`.

You may be wondering at this point how the `JSR` instruction differs from the `JMP` instruction. The difference is that the `JSR` saves the address of the instruction following it somewhere (we'll talk about

```
; Main program which calls the multiply subroutine to multiply
; two numbers, mpee and mpor.

280040    ;0000:  mov r0, mpee    ;r0 contains the multipliee
290042    ;0003:  mov r1, mpor    ;r1 contains the multiplior
2D0100    ;0006:  jsr mult        ;Jump to subroutine
00        ;0009:  halt
.ORG 0040
    ;mpee:
00a3      ;0040:  multipliee
    ;mpor:
002c      ;0042:  multiplior

; Subroutine to multiply two numbers passed as arguments in
; r0 and r1.  The result is returned in r2.  The contents of
; all registers modified during execution of the subroutine
; are restored before returning.

.ORG 0100
    ;mult:
240123    ;0100:  mov stor0, r0       ;Save r0, r1, and r3
250125    ;0103:  mov stor1, r1
270127    ;0106:  mov stor3, r3
220000    ;0109:  mov r2, #0000       ;Initialize accum. sum
    ;start_loop:
230001    ;010c:  mov r3, #0001
9d        ;010f:  and r3, r1          ;Is l.s.b. zero?
510114    ;0110:  br 1, bit_is_zero   ;Branch if yes
78        ;0113:  add r2, r0          ;add r0 to sum if not
    ;bit_is_zero:
88        ;0114:  lshift r0
8d        ;0115:  rshift r1
52010c    ;0116:  br 2, start_loop    ;Do again if r1 != 0
280123    ;0119:  mov r0, stor0       ;Restore r0, r1, and r3
290125    ;011c:  mov r1, stor1
2b0127    ;011f:  mov r3, stor3
2f        ;0122:  rts                 ;Return
    ;stor0:
0000      ;0123:  Storage for r0
    ;stor1:
0000      ;0125:  Storage for r1
    ;stor3:
0000      ;0127:  Storage for r3
```

Figure 15−12. Modification of program in Fig. 15−11 to convert it to a subroutine. Also included is a short program which calls the subroutine.

where shortly), so that the subroutine can know where to return when done; whereas `JMP` does not. If the instruction at address `0006` in the main program had been `jmp mult`, the program would have executed the code in the subroutine with no problem, but when finished, the `rts` instruction at address `0122` would almost certainly send it to some strange place because the correct address for the return would not have been stored by the `jmp mult` instruction.

Let's now discuss the subroutine. The multiplyee and multiplior are to be passed to the subroutine in registers `R0` and `R1`, respectively, and the answer is returned in `R2`. In order to carry out the multiplication, it will be necessary to use (and therefore modify the contents of) registers `R0`, `R1`, and `R3`. If the subroutine is to be treated as a "black box," the inner workings of which need not be remembered, using it should not have any side effects other that those expected (modifying the contents of `R2` to contain the answer in this case). If we aren't careful, the contents of `R0`, `R1` and `R3` will also be modified on return. That's a

mistake just waiting to happen, so I've added some code at the start of the routine to save the contents of these three registers so that they can be restored when the routine is finished. To do that, six bytes of memory are set aside at the end of the subroutine for storage, and the contents of the three registers are MOV'ed there. At the end of the routine, the contents of these memory locations are MOV'ed back to the registers.

Besides this change and some obvious address changes, the only change to the original multiply routine is the replacement of the HALT instruction with a RTS instruction. This latter instruction (ReTurn from Subroutine) causes execution to continue with the instruction following the JSR instruction that originally called the subroutine.

Running the program with the emulator produces the same result in R2 as previously. Fig. 15−13 shows the first five and last four lines of the report file. Notice how the execution address changes just after the JSR and RTS instructions. (For the moment, ignore the stuff about SP in the lines involving the JSR and RTS instructions. It's related to where the JSR instruction stores the return address, and I'll get to that shortly.)

```
0000:  MOV R0, 0040 :  280040  ;  R0=00A3, CR=000A
0003:  MOV R1, 0042 :  290042  ;  R1=002C, CR=000A
0006:  JSR 0100     :  2D0100  ;  SP -> 0FFE, @(SP) -> 0009, PC:  0006 -> 0100
0100:  MOV 0123, R0 :  240130  ;  @(0130)=00A3, CR=000A
0103:  MOV 0125, R1 :  250132  ;  @(0132)=002C, CR=000A

         .                .              .

         .                .              .

         .                .              .

011C:  MOV R1, 0125 :  290132  ;  R1=002C, CR=000A
011F:  MOV R3, 0127 :  2B0134  ;  R3=0000, CR=0001
0122:  RTS          :  2F0000  ;  SP -> 1000, PC:  0122 -> 0009
0009:  HALT
```

Figure 15−13. First five and last four lines of the report file for the program in Fig. 15−12.

The JSR instruction in the third line of the report causes the CPU to jump to the instruction at 0100, the starting address of the multiply subroutine, and to save the address of the instruction following the JSR (0009 in this case) somewhere. The RTS instruction in the next-to-last line of the report retrieves this return address and causes the CPU to jump to it.

15.6 Stacks

I can use this subroutine example to illustrate another very common programming technique—the use of a *stack*. Often a programmer needs some memory space temporarily. For example, in the subroutine in Fig. 15−12 we needed space to save temporarily the contents of three registers. When the subroutine is finished, this space is no longer needed. It is wasteful of memory space to reserve six bytes solely for this use. For this and other (better) reasons, almost all programs use what is called a stack to allocate temporary memory space. Here's the idea. A register or memory location is initially loaded with the highest address of some block of unused memory. Typically, this address would be the highest address available to the program. The register containing this address is called the *stack pointer*, and the address contained in it is called the *top of the stack*. Whenever the programmer wants to store a number temporarily, the stack pointer is decremented to point to the next free space, and the number is MOV'ed to the address contained in the stack pointer. This place on the stack (the position of the last number put on the stack) is called the *top of the stack*. (Actually, it probably should be called the *bottom* of the stack, but it isn't.) Decrementing the stack pointer and MOV'ing the number in this way is referred to as *pushing the number onto the stack*.

The term *stack* comes, I think, from an analogy with a literal stack. When you want to save some numbers, you put them on the top of the stack. If you have several numbers to save, then the stack consists of a pile of numbers, one on top of the other, with the first one put onto the stack on the bottom, and the last on the top. It's like a stack of trays in a cafeteria. When you want the numbers back, you take them off of the stack. If you were clever, and put the numbers onto the stack in the reverse order of the order you need to get them back, then the operation is easy. You are always putting numbers onto the top of the stack, and taking them off of the top as well. If you need to retrieve a number other than the one on the top of the stack, things are a little trickier. You have to remember where it is in the stack, and then take it out without messing up the other numbers on the stack.

If the number you want to retrieve from the stack is the number on the top, the operation of retrieving it is called *popping the number off of the stack*. To do that with the computer stack, you `MOV` the number at that address contained in the stack pointer to a register, and increment the stack pointer by the length in bytes of the number so that the stack pointer still points to the number on the top of the stack. Besides retrieving the number, this process also returns the memory used for storing the number back to the stack for later use. (Actually, the number is still there, but it can be overwritten by a subsequent push operation, so it has become a kind of zombie, neither alive nor dead.) If the number you want was not the one you last pushed onto the stack, things are a little more complicated. You have to know how far down it is on the stack and form its address by adding the corresponding number of bytes to the stack pointer. The number can then be retrieved using this address and a `MOV` instruction. The number still occupies space on the stack after this procedure, and some method must be used to return this space when finished with it. This latter operation is referred to as *cleaning up the stack*.

The use of stacks is so common that most CPU's set aside a special-purpose register to contain a stack pointer, and the instruction set of these CPU's includes instructions to push numbers to the stack and pop numbers from it. In the PFW007, the stack pointer register is called `SP`, and the push and pop instructions are `PUSH Ra` and `POP Ra`. The `PUSH Ra` instruction subtracts two from `SP` and then copies the number in the register specified by the operand `Ra` to the memory location pointed to by `SP`. The `POP Ra` does the opposite. It copies the number at the address pointed to by `SP` to the register specified by the operand, `Ra`, and then adds two to the contents of `SP` so that it still points to the top of the stack.

There also are instructions for moving numbers to and from the `SP` register. `MOV SP, Rb` and `MOV SP #Adr` load the `SP` register with a specified number. The first loads it with whatever `Rb` contains, and the second with the number specified by `Adr`. The `MOV Ra, SP` instruction does the reverse, it loads the register specified by `Ra` with the contents of `SP`. The operation of these three instructions is exactly analogous to that of the first two `MOV` instruction in Table 14−1, except that one register is `SP`.

To use a stack, one must first initialize the stack pointer with the address of the highest byte of memory available to the stack (usually `0FFFH` for the PFW007).[2] The `MOV SP, #Adr` or perhaps the `MOV SP, Ra` instruction can be used for this purpose. If you want to initialize `SP` to the highest available address, `0FFFH`, you actually don't have to do anything because the emulator automatically initializes the contents of `SP` to this value when you start it. All other registers and all memory locations are initialized to zero. To clarify how the stack operations work, Table 15−2 shows what happens to the contents of several registers and memory locations after a sequence of such operations. The values in the table are the values *after* the instruction in the left-hand column is executed.

In our multiply subroutine, we want to save temporarily the contents of registers `R0`, `R1`, and `R3`. Instead of setting aside six bytes of memory specifically for this storage, we might use the stack. At the start of the subroutine, we could push the contents of `R0` onto the stack, followed by `R1`, and then `R3`. At the end of the subroutine, we could restore the values of these three registers, as well as clean up the stack

---

2. There is a small mud puddle here. Because `PUSH` operates by first decrementing `SP` and then `MOV`'ing the number to the resulting address, the stack pointer should be set to the highest address you want to use *plus 2*. Initializing `SP` to `0FFFH` wastes two bytes of memory, but the emulator does so anyway to avoid confusion (I hope!).

Instruction	R1	R2	SP	0FFD	0FFB	0606	0604
	0111	0222	1234	0000	0000	0000	0000
mov sp, #0FFF	0111	0222	0FFF	0000	0000	0000	0000
push r1	0111	0222	0FFD	0111	0000	0000	0000
add r1, r2	0333	0222	0FFD	0111	0000	0000	0000
push r1	0333	0222	0FFB	0111	0333	0000	0000
pop r2	0333	0333	0FFD	0111	0333	0000	0000
pop r1	0111	0333	0FFF	0111	0333	0000	0000
mov sp, #608	0111	0333	0608	0111	0333	0000	0000
add r1, r2	0444	0333	0608	0111	0333	0000	0000
add r2, r1	0444	0777	0608	0111	0333	0000	0000
push r1	0444	0777	0606	0111	0333	0444	0000
push r2	0444	0777	0604	0111	0333	0444	0777
pop r1	0777	0777	0606	0111	0333	0444	0777
pop r2	0777	0444	0608	0111	0333	0444	0777
mov r1, sp	0608	0444	0608	0111	0333	0444	0777
mov sp, r2	0608	0444	0444	0111	0333	0444	0777

TABLE 15−2. Table showing the effect of a sequence of stack-oriented instructions on the contents of several registers and memory locations. The values of these registers and memory locations shown are the values *after* the instruction in the first column is executed. The instructions are assumed to be executed in sequence. The horizontal lines in the table are intended only as a guide to the eye.

by simply popping them off the stack in the reverse order to that used to push them onto the stack. Notice that the order the registers are popped off the stack is important. The wrong order will scramble the contents of the registers.

The program in Fig. 15−14 shows the modified subroutine which saves the registers onto the stack. The modified subroutine and the main program that calls it are similar to the previous version. The main program is the same except that an instruction is added (`mov sp, #0fff`) which initializes the stack pointer to point to the top of the memory available. (As I mentioned previously, this instruction is not really required since the emulator does this anyway, but I wanted to show the initialization of the stack pointer explicitly.) In the subroutine, the three `MOV` instructions that were used to save the values of `R0`, `R1`, and `R3` are changed to `PUSH` instructions, and at the end, the three `MOV` instructions that were used to restore the values of these registers were changed to `POP` instructions. The only other changes were changes to the addresses caused by the addition of the `MOV` instruction to the main program, and the different lengths of the `PUSH` and `POP` instructions.

15.7 Subroutines and Stacks

Stacks are used frequently in subroutines. In this section I want to discuss some of these uses, and to expand a little more on the use of subroutines. Unless you become a professional programmer, it is unlikely that you will make use of these techniques directly in assembly language programming, but most high-level language compilers use these techniques to produce a machine language program equivalent to the high-level program. You really don't have to be very adept at using these techniques because the compilers do most of the work for you, but it is helpful to have a general idea about what is happening.

15.7.1 The Return Address

At the start of section 15−5, when I introduced the `JSR` instruction, I mentioned that the return address is stored somewhere, but I did not tell you where. Now I can reveal all. The return address is usually pushed onto the stack. Instead of this approach, one could reserve two bytes of memory not associated

```
; Program to multiply two numbers using the multiply subroutine
; modified to use a stack

280040    ;0000:  mov r0, mpee    ; r0 contains the multipliee
290042    ;0003:  mov r1, mpor    ; r1 contains the multiplior
2e0fff    ;0006:  mov sp, #0fff   ; Initialize stack pointer
2D0100    ;0009:  jsr mult        ; Jump to subroutine
00        ;000c:  halt
.ORG 0040
    ;mpee:
00a3      ;0040:  multipliee
    ;mpor:
002c      ;0042:  multiplior

; Subroutine to multiply two numbers passed as arguments in
; r0 and r1.  The result is returned in r2.  The contents of
; all registers modified during execution of the subroutine
; are saved on the stack and restored before returning.

.ORG 0100
    ;mult:
                                    ; Save r0, r1, and r3
60        ;0100:  push r0
61        ;0101:  push r1
63        ;0102:  push r3
220000    ;0103:  mov r2, #0000     ; Initialize accum. sum
    ;start_loop:
230001    ;0106:  mov r3, #0001
9d        ;0109:  and r3, r1        ; Is l.s.b. zero?
51010e    ;010a:  br 1, bit_is_zero ; Branch if yes
78        ;010d:  add r2, r0        ; Add r0 to sum if not
    ;bit_is_zero:
88        ;010e:  lshift r0
8d        ;010f:  rshift r1
520106    ;0110:  br 2, start_loop  ; Do again if r1 != 0
                                    ; Restore r0, r1, and r3
67        ;0113:  pop r3
65        ;0114:  pop r1
64        ;0115:  pop r0
2f        ;0116:  rts               ; Return
```

Figure 15−14. Modification of multiply subroutine to use a stack for allocating memory to temporary variables.

with the stack specifically for storing this address, but then a subroutine could not call another subroutine because calling the second subroutine from inside the first would overwrite the return address for the first with that for the second. The first would then not have the correct return address, and the program is sure to go astray.

If, on the other hand, the stack is used for this purpose, there is no such problem because the return address for each subroutine just gets pushed onto the stack. Each subroutine call gets its own stack space for a return address. A disadvantage of this approach is that it makes it easier for the programmer to go astray. One must be careful to restore the stack just before exiting the subroutine to the condition it had just after entering it, or the return address will get all messed up. On the PFW007 the JSR instruction automatically pushes the return address onto the stack, and the RTS instruction automatically pops this address off the stack into the program counter register.

15.7.2 Passing Arguments to Subroutines

Another use of the stack is to pass information to subroutines. In the multiplication subroutine, the two numbers to be multiplied were passed to it in registers `R0` and `R1`. These numbers are called *arguments*. The usage of the term here is in good analogy with the use of the term when referring to the arguments of a function. In both cases, the argument passes the information needed for the subroutine or function to determine a value to return. Because there are a limited number of registers, the stack is often used in place of registers to pass arguments to subroutines. The arguments are simply pushed onto the stack just before calling the subroutine, and the subroutine accesses them as needed. This is the convention that C uses. The bookkeeping required to remember where each argument is on the stack, and to clean up the stack after the subroutine returns is simple but tedious. Fortunately, compilers for higher level languages such as C take care of all these details without the programmer having to know much about them, and attending to simple but tedious tasks is something computer programs such as compilers are very good at.

15.7.3 Recursion

In the example of the multiply subroutine, we needed to use the three registers `R0`, `R1`, and `R3` inside the subroutine. Since these registers could contain numbers which are needed in the main program, it was important that the original values be saved at the start of the subroutine so that they could be restored at the end of the subroutine. In the second version of this program, we used the stack for storing these values temporarily. The reason I gave for using the stack rather than permanent storage was efficiency of memory usage: it wastes space to reserve space for the variables even when the subroutine is not being executed. This is a correct, but not very convincing reason. A more important reason to use the stack rather than permanent storage is that when the stack is used the subroutine can call itself. If permanent storage is used (as in the program in Fig. 15−10) and the subroutine calls itself, the stored values from the second call will overwrite those of the first call and things will get messed up. Using a stack, on the other hand, each call to the subroutine gets its own space on the stack.

A subroutine that calls itself is said to be *recursive*. It might seem that such a feature has little value, but in some cases recursion simplifies programs considerably. The last section of this chapter describes a subroutine which uses recursion to calculate the factorial of a number. Recursion is kinda nifty, but you can really get in trouble with it! One little teensy mistake, and the subroutine keeps calling itself. Each call uses more space on the stack, until all the memory allocated for the stack is used up. If the system is well protected it bombs out right away. If not, it continues merrily overwriting other stuff in memory. If that stuff was important, the computer goes bonkers, and you have to reboot it. When a program runs out of stack space, it is rather appropriately said to have *blown its stack*.

15.7.4 Automatic and Static Storage

Programs manipulate numbers, called *variables*. So far, the programs we have written have been pretty simple, and the four registers of the PFW007 were sufficient to hold all the variables involved. More commonly, programs are much more complex and space is needed to hold many more variables than there are registers available. In this case, the variables must be stored in memory and moved to and from registers as needed. How should memory space for these variables be assigned?

For variables used in the main program, a good method would be to assign memory space permanently to each variable. For variables used in a subroutine, however, this method wastes space, and does not allow recursion. If a particular variable is needed only temporarily a better method is to get storage space from the stack. The storage space can be used while needed, and then returned to the system. A variable assigned permanently to a specific memory location is said to be assigned *static* storage, and the variable is often called a *static variable*. In this context, the term *static* does not mean that the value of the variable cannot change, but rather that the address used to store the variable can't. A variable assigned space from the stack is said to be assigned *automatic* or *volatile* storage, and the variable is sometimes called an *automatic* variable. It "automatically" appears and disappears as needed.

Automatic storage is frequently used for variables inside of subroutines. This space is needed only while the subroutine is being executed, and can be safely returned to the system for other uses when the

subroutine returns. Reserving automatic space is easy—just decrement SP by the amount of space needed. If there is more than one automatic variable, you also have to remember where each one is stored on the stack. When the subroutine is finished, the memory is returned to the pool by simply restoring SP to the value it had on entry to the subroutine. The whole process requires some careful bookkeeping, but it's simple in principle. The compiler programs for most higher level languages such as C use such a technique for allocating memory for automatic variables. Fortunately, the compiler takes care of all the tedious bookkeeping so that the programmer does not need to worry about it.

15.8 Reading a Number from the Input File

I'll end this chapter with two examples of programs which use subroutines in realistic ways. In this section I'll write a program that reads a number from the input file and puts it into memory. This example illustrates a realistic application of the multiplication subroutine we developed. In the next section, I'll write a program which contains a recursive subroutine that calculates the factorial of a positive integer.

At first glance, a program to read a number from the input file and put it into memory might seem simple; you just use a `MOV R?, F000` instruction to put the number into some register (`R?`), and then move the number to memory with another `MOV` instruction. This approach does not work, however, because the number in the file is to be represented in human-convenient form, which is as a string of ASCII characters giving the base ten representation of it. For example, the number 1396 would be in the file as four bytes, one for the ASCII code for each character, `31H 33H 39H 36H` to be exact. The program has to convert such a character string to the actual number 1396. That takes a little more doing.

The first thing we need to do is to design the algorithm we are going to use. The number 1396 equals $1\times1000 + 3\times100 + 9\times10 + 6$. I could make an algorithm which works that way, using my newly fashioned multiplication subroutine to do the multiplications. The problem is that the first digit I read in the file is the 1, and at the time I read it I don't know what power of ten to multiply it by. I only know that after I've read all the digits and counted how many there are. One solution to the problem would be to push the individual digits onto the stack as they are read, and then when all the digits are read from the file to pop them off the stack, multiplying them by the proper factor of 10 as we go.

Here's another, I think simpler, way. I'll use one register, say `R0`, to hold the sum as it accumulates. The register will be loaded initially with zero. I'll read the first character from the input file, and if it's the ASCII code for a legal digit I'll add the value it represents to `R0`. If it's not a digit, I'll jump to the end of the program, otherwise I'll then read the next character from the file, and if it's a digit, I'll multiply the contents of `R0` by 10 and add the value the new digit represents to the result. I'll continue this process until I read a character that's not a digit, whereupon I'll assume I've read the whole number and jump to the end of the program. I could express my algorithm mathematically by

$$1396 = 10\times(10\times(10\times(1) + 3) + 9) + 6$$

Fig. 15−14 shows a flow chart of the general idea. I added a check to see if we had reached the end of the file (EOF) while reading in digits.

To implement the algorithm shown in Fig. 15−15, I have several sub-problems to solve. First, how do we convert the ASCII code for a digit to the value it represents? When I read the first digit, I place the ASCII code for the character `1`, not the number 1 itself in `R3`. Thus `R3` contains `0031H`, and I want it to contain `0001H`. There's a similar problem for the other digits, so the method must work for all the base-ten digits. Looking at the table of ASCII codes in hex, I see a solution. The codes for the digits `0` through `9` run from `30H` through `39H`, so I can convert the ASCII code for a digit to the number it represents by just subtracting `30H` from it. The PFW007 doesn't have a subtract instruction, but I can subtract by taking the two's complement of `30H` and adding it. The 16-bit version of the two's complement of `30H` is `FFD0H`.

A second problem is how to determine whether or not a character is a legal digit. If the character is less than `30H` or greater than `39H`, it ain't kosher, so I can test the legality of a character by using the `CMP` instruction to compare the character with `30H` and with `39H`. Still a third problem is how to multiply the

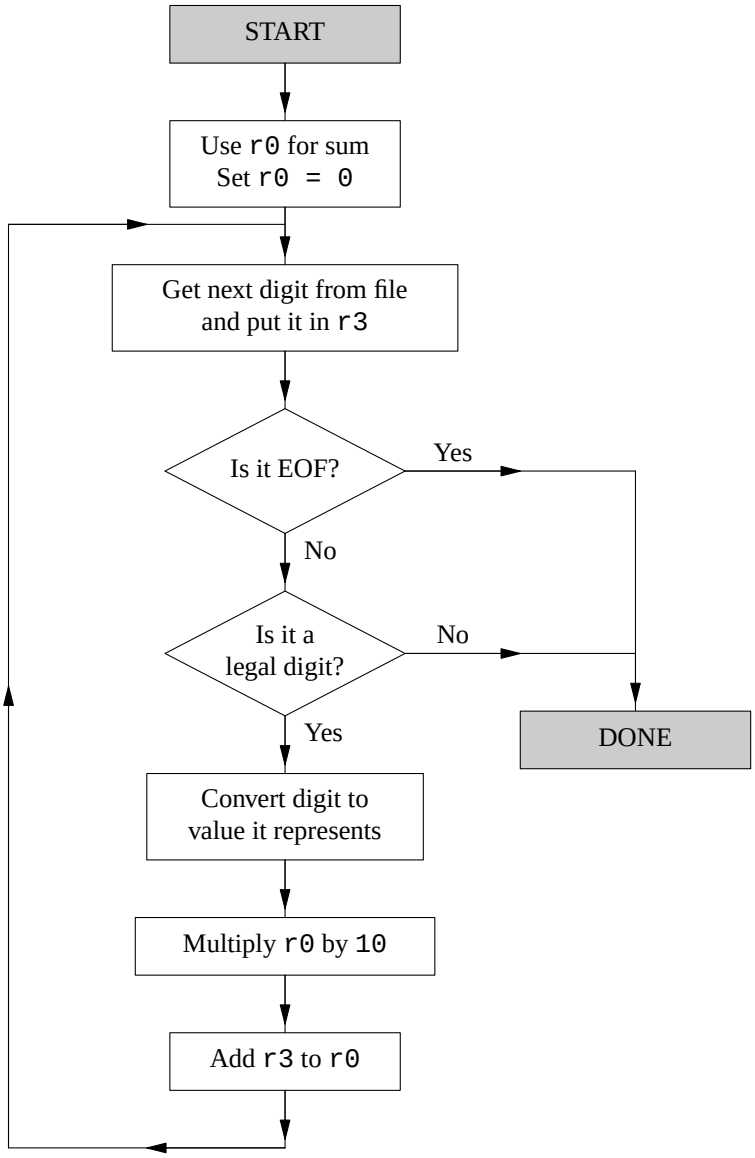

Figure 15−15. Flow chart for the algorithm discussed in the text for reading an ASCII-coded number from a file and convering it to the value it represents.

contents of register `R0` by `10`. This one I've already solved by writing the subroutine in Fig. 15−14, so all I need to do is put the multiplyee and multiplior in registers `R0` and `R1`, and do a `JSR` to the multiply subroutine. The subroutine returns the result in `R2`, so I will need to `MOV` this value into `R0`. The `R0` register already contains the multiplyee, so to prepare for the subroutine call, I need only load `R1` with `000AH` (that's 10 expressed in hex).

With these preparations, it's just a matter of writing the machine code. The program is shown in Fig. 15−16. To use the program you have to copy the stack version of the multiply subroutine onto the end of it. You also have to use an editor to create a file with a number to be read in, like `1396`, in it. The `1`

```
; Program that reads one line from input file, assumes it
; to be the decimal representation of a number (in ASCII),
; and converts it to a number which is placed in r0.
; To run the program, the multiply subroutine in
; Fig. 15-12 must be copied at the end, starting at
; memory location 0100H.

200000    ;0000:  mov r0, #0       ; r0 holds accum. sum
    ;next:
2bf000    ;0003:  mov r3, f000     ; get a char
540024    ;0006:  br 4, end        ; Check for EOF
                                   ; Is char a legal digit?
210030    ;0009:  mov r1, #0030
cd        ;000c:  cmp r3, r1
540024    ;000d:  br 4, end        ; No if r3 < 30H
210039    ;0010:  mov r1, #0039
cd        ;0013:  cmp r3, r1
580024    ;0014:  br 8, end        ; No if r3 > 39H
                                   ; Convert from ASCII
21ffd0    ;0015:  mov r1, #ffd0    ; -'0'
7d        ;0018:  add r3, r1
                                   ; Multiply sum by 10
21000a    ;0019:  mov r1, #a       ; Sum is already in r0
2d0100    ;001c:  jsr mult         ; Multiply sum by 10
12        ;001f:  mov r0, r2       ; Move result to r0
73        ;0020:  add r0, r3       ; Add last digit to sum
2c0003    ;0021:  jmp next         ; Do it again
    ;end:
00        ;0024:  halt

.ORG 0100
; Multiply subroutine goes here.
```

Figure 15−16. Main program to read a character string from the input file and convert it to the number it represents. The program uses the stack version of the multiply subroutine in Fig. 15−14, and this subroutine must be copied at the end as indicated.

must be the first character in the file. The file must have the same name as your program file, except that the extension must be `.IN`. When finished, the result should be in register `R0`. I ran the program with 1396 in the input file, and got

```
R0=0574   R1=0030   R2=056E   R3=000A
```

The contents of `R0` are `0574`. That's 1396 expressed in hex, so the program worked!

This program is the first realistic example of the use of a "canned" subroutine I wrote previously to be used as a tool. In writing the program to convert the text representation of a number to the actual number we had enough to worry about just trying to figure out how to do it, without having to worry about the multiplication stuff. Writing the multiply subroutine was quite a bit of trouble, but once it's done and tested we can just use it without thinking very much about what it's doing. That is a big help.

Actually, it would make sense to convert the main program itself to a "black box" subroutine. Reading a number from a file is a common task, and a subroutine to accomplish that would be useful. Instead of doing it here, Ill leave it to an exercise at the end of the chapter. I won't go into it here, but there is a similar need for a subroutine to convert a number stored internally to the equivalent string of ASCII characters so that it can be printed from a program. This is the inverse of the previous problem. The only algorithms I

know of to solve it require dividing by the base of the number system you want to use to print out the number (10, for example). The PFW007 does not have such an instruction, and we have not written a division subroutine, so I won't attempt to write such a program. Division by a power of two can be accomplished by doing the proper number of left shifts, however, so you might try writing a program to print to a file the value of a number expressed in octal (base 8).

I'm not the first person to realize that such a set of subroutines would be handy. In fact, there is a subroutine supplied with all C compilers and called `scanf` which accomplishes the task of converting a number expressed as an ASCII string to a number stored internally to the computer. There is another standard subroutine, called `printf`, which accomplishes the inverse task of converting a number stored internally to a set of ASCII characters that can be printed. There routines are much more sophisticated than the program in Fig. 15−16, and they can convert to and from floating point representations of the number as well as integer, and you can specify that the ASCII representation of the number can be octal, hexadecimal, or decimal (base 10).

15.9 Recursion and a Program to Calculate n!

As an example of the use of recursion, we will write a subroutine that calculates the factorial of a number passed to it as an argument. Let's call the value of the argument *n*, and the subroutine will be written so that it returns the value of *n!*. I'll write it so that the argument is found in register `R0` on entry to the subroutine, and the value of *n!* is placed in `R2` on return.

Here's how the subroutine will work. It will use directly the definition of the factorial of a positive integer: $n! = n \cdot (n-1)!$ for $n > 1$, and $1! = 1$. On entry, the subroutine will first check to see if its argument contained in register `R0` is 1 or not. To simplify the discussion, lets call the value of this argument a. If a is not 1, the subroutine will get the value of $(a-1)!$ by calling itself with an argument of $a-1$. When this call returns, it will multiply the value the call produces by the value of a to form the product $a \cdot (a-1)!$, as per the definition. The subroutine will then return the value of this product by placing it in register `R2` and returning. If, on the other hand, n is 1, the subroutine simply returns the value 1 (as per the definition of $1!$) by simply placing the number 1 in register `R2` and then returning. Fig. 15−17 shows a flow chart of the subroutine I have in mind.

As an example Fig. 15−18 shows a diagram of how the subroutine will work when called with an argument of 4. The initial call results in the subroutine calling itself three more times, first with argument 3, then with argument 2, and finally with argument 1. Each time the subroutine is called it checks to see if its argument equals 1. The first three times the argument is greater than one, so it just calls itself with an argument that is one less than what it was called with and waits for the call to return with a value. On the fourth call, the argument is equal to 1, so this call can return immediately with value 1. Execution returns to the previous call which sees 1 as the return value for its $(a-1)!$. It multiplies this value times its own argument, 2 in this case, places the result in register `R2` as a return value, and returns execution to the call just above it. The process repeats until finally the top call returns with the desired value ($4 \times (3!)$ in this case).

In order to test the subroutine, we will need a main program which will call the subroutine. Fig. 15−19 shows the subroutine and a simple main program that calls it with an argument of 4. The main program is very easy. It simply puts the number to be "factorialed" (4 in the case shown) into register `R0` where the subroutine expects to find it and calls the subroutine `fact`. Once you understand the general strategy, writing the subroutine is also pretty simple. The subroutine needs a register to use for internal computation. I chose to use `R1` for this purpose, so the first thing the subroutine does is save the contents of `R1` by pushing it onto the stack. Labeling the argument a as above, the routine then checks (in the next three instructions) to see whether or not a is one. If $a = 1$, the branch is not taken, and the number 1 is MOV'ed to register `R2`, register `R1` is restored, and the subroutine returns.

If $a \neq 1$, the subroutine should return the value $a \cdot (a-1)!$. In this case the branch to `not_done` is taken, and the subroutine prepares to call itself with argument $a-1$ in order to obtain the value of $(a-1)!$.

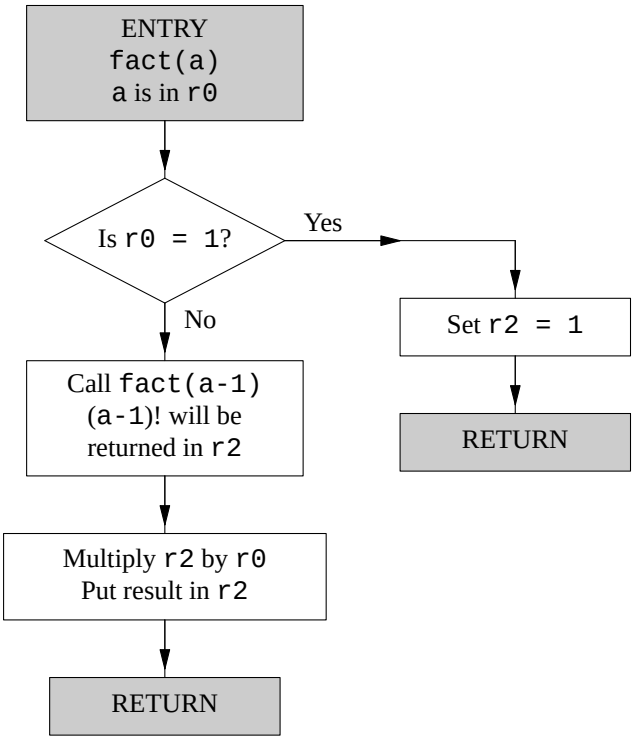

Figure 15−17. Flow chart for a subroutine that calculates the factorial of its argument.

To do that, it will have to subtract 1 from the value in `R0`. That will modify the contents of `R0`, so the subroutine first saves the unmodified value for later restoration. The routine then decrements the value in `R0`, and calls `fact`. Since `fact` expects to find its argument in this register, the factorial of this value should be in `R2` when the call returns. The subroutine copies this return value into `R1`, restores the original value of a to `R0`, and calls the multiply subroutine, `mult`. This subroutine multiplies the values it finds in `R0` and `R1`, and places the result in `R2`. This value should be just $a \cdot (a-1)!$, the desired value, and it's in the correct register, so `fact` simply restores `R1` and returns.

After copying the multiply subroutine from our previous examples onto the end of this program, it runs and produces

```
R0=0004   R1=0000   R2=0018   R3=0000
```

The value of 4! should appear in register `R2`. At first glance, it appears that the program did not work correctly, because $4! = 24$, not 18. But wait! One of the secrets to success in life is to avoid panicing too soon. The emulator uses hexadecimal, and the values in the registers are written in hex, so the value in `R2` in base 10 is $1 \times 16 + 8 = 24$. The subroutine did, in fact, work.

15.10 APPENDIX: Using the Turbo C Editor

A pretty good editor is included as part of the Turbo C compiler package. The purpose of this appendix is to give you enough information about the editor to use it. I'm not an expert, and much of what is here I got from the manual.

To start the package, simply enter `tc` at the DOS prompt. After a delay you should get a screen with

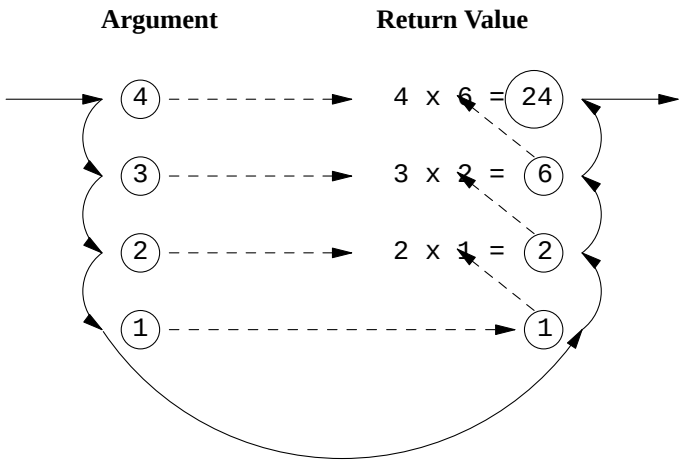

Figure 15−18. Schematic flow diagram showing the operation of the recursive subroutine calculating the value of 4!. The solid arrows show the flow of control, and the dashed arrows show the flow of data. Flow starts at the upper left and the subroutine is entered with argument 4. It then calls itself three more times in sequence with arguments of 3, 2, and 1. Finally the chain of calls starts returning, as shown by the flow control arrows on the right side, providing the factorial values shown in the circles on the right. At each stage, the value in the right-hand circle should be the factorial of the value of the argument in the left-hand circle on the same line.

a menu bar the looks something like the Quattro Pro menu bar at the top, and who knows what below. (The program seems to have a memory about what was being done previously, and it tries to get back in the same state it was.) If the middle of the screen has anything in it, get rid of it by pressing the **Alt** and the **F3** keys simultaneously. Keep doing it until the screen is pretty much blank. The jargon for what you just did is you *closed the windows*.

The Turbo C package has a number of capabilities. It's sold by the same company that sells Quattro Pro, so the operation is somewhat similar. To access some feature you must activate the menu bar, similarly to Quattro Pro, the only difference is that you use the **F10** key, rather than the slash, /. To edit a file access the **File** menu. Do so by pressing **F10** and then **F**. You can also use the arrow keys to move the highlight, or you can use the mouse. Doing that gives a menu with several items. Choose **Open** and you get a box asking for the name of the file. Type it in and press **Enter**. A window will appear, with the name you just typed in at the top. (If you did not specify an extension, **tc** appends `.C` to it.) Anything you type will appear on the screen, and when you save what you have done, it will be placed in the file with the given name. You save the file with the **Save** item of the **File** menu.

The rest of this section describes some of the features of the editor part of the **tc** package. Table 15−3 summarizes some of the editor commands. Once you have an edit window open, there is a blinking cursor. When you type a character it appears just before the cursor. When you press **Enter**, the cursor goes to the start of the next line. In many other ways, the editor works similarly to Quattro Pro in the Edit mode. The **Delete** key deletes the character under the cursor, **Backspace** the character just to the left. **^T** deletes the word to the right of the cursor, and **^Y** deletes the entire line. You can move the cursor around with the arrow keys. The → and ← keys move one character, and ⇑ and ↓ move one line. ^→ and ^← move one word. **Home** moves to the start of a line, **End** to the end, and **^Home** and **^End** moves to the top or bottom of the screen. You go to the next screenful with **PgUp** and **PgDn**, and **^PgUp** and **^PgDn** moves to the start and end of the file.

```
; Program to calculate the factorial of a number, n, using a
; recursive subroutine.

280020    ;0000:  mov r0, n       ; Get number to be "factorialed"
2d0050    ;0003:  jsr fact        ; Calculate n!
00        ;0006:  halt

.ORG 0020
        ; n:
0004      ;0020:  n               ; Storage location for n

; Recursive subroutine to calculate a factorial.
; On entry, the number to be "factorialed" should be in
; register r0, and on return the answer is left in r2.
; The routine does not check for illegal arguments or for
; overflow of the result.  The subroutine requires the
; multiply subroutine loaded at address 0100.

.ORG 0050
        ; fact:
61        ;0050:  push r1         ; Save r1
                                  ; Is argument (a) 1?
210001    ;0051:  mov r1, #0001
c1        ;0054:  cmp r0, r1
58005d    ;0055:  br 8, not_done  ; Branch if a > 1.
220001    ;0058:  mov r2, #0001   ; Else, put 1 in r2,
65        ;005b:  pop r1          ; restore r1, and
2f        ;005c:  rts             ; return.
    ; not_done                   ; Get here only if a > 1
60        ;005d:  push r0         ; Save r0
84        ;005e:  dec r0          ; Subtract 1 from a
2d0050    ;005f:  jsr fact        ; Calculate (a-1)!
                                  ; (a-1)! is returned in r2,
16        ;0062:  mov r1, r2      ; Put it in r1 for mult.
64        ;0063:  pop r0          ; Restore a to r0
2d0100    ;0064:  jsr mult        ; Multiply a times (a-1)!
                                  ; Result is already in r2,
65        ;0067:  pop r1          ; restore r1, and
2f        ;0068:  rts             ; return.

.ORG 0100
; Multiply subroutine goes here.
```

Figure 15−19. Program using a recursive subroutine to calculate the factorial of a positive integer. This program requires the multiply subroutine which is assumed to have been loaded at address `0100H`.

You can move text around by first marking it, then **Cut**ing or **Copy**ing it to the Clipboard, moving the cursor to where you want the text placed, and then **Paste**ing it. You mark text by moving the cursor to the start of the block you want to mark, then pressing the **Shift** key and using the arrow keys. The marked text is highlighted. If you want to copy the block (so there will be two copies of the same material in the file) then choose the **Copy** item from the **Edit** menu. If you want to move it (so there will be only one copy) choose the **Cut** item of the **Edit** menu. After doing this, move the cursor to where you want to text to be placed, and choose the **Paste** item of the **Edit** menu.

A useful feature of the editor is the capability of finding a character string in a file and, optionally, of replacing it with another string. To just find a string, choose the **Find** item of the **Search** menu, and type the desired string into the resulting box. The cursor should move to the next occurrence of the string. If you want to search again for the same string (to find the next occurrence) press **^L** or choose **Search Again**

SOME TURBO C EDITOR COMMANDS

Action	Key(s)
Move by 1 char.	←, →
Move by 1 word	^←, ^→
Move by 1 line	⇑, ↓
Move to begin of line	**Home**
Move to end of line	**End**
Move to window top	**^Home**
Move to window bottom	**^End**
Move 1 page up	**PgUp**
Move 1 page down	**PgDn**
Move to begin of file	**^PgUp**
Move to end of file	**^PgDn**
Delete char to left of cursor	**Backspace**
Delete char at cursor	**Del**
Delete word to right of cursor	**^T**
Delete line	**^Y**
Delete to end of line	**^QY**
Insert line	**^N**
Go to menu bar	**F10**
Get out of menu	**Esc**
Quit the editor	**File|Quit** or **Alt-X**
Open a file	**File|Open** or **F3**
Save to a file	**File|Save** or **F2**
Undo last change	**Edit|Restore Line** or **^QL**
Mark block	**Shift** ←, **Shift** →, **Shift** ⇑, **Shift** ↓
Move block	**Edit|Cut, Edit|Paste** or **Shift-Del, Shift-Ins**
Copy block	**Edit|Copy, Edit|Paste,** or **^Ins, Shift-Ins**
Delete block	**Edit|Clear** or **^Del**
Search	**Search|Find** or **^QF**
Search and replace	**Search|Replace** or **^QA**
Repeat last search	**Search|Search Again** or **^L**

TABLE 15−3. Some Turbo C editor commands.

from the **Search** menu. If you want to replace the searched-for character string with another string, choose the **Replace** item of the **Search** menu. After you have specified the two strings (the first giving the string to be searched for and the second the string to replace it with), the editor will highlight the first occurrence of the searched-for string and ask you if you want to replace it. Respond with either yes or no. You can use **^L** then to go to the next occurrence of the search string for optional replacement.

When you are done editing the file, be sure to **Save** it. You can then get back to DOS by either pressing **Alt-X**, or choosing the **Quit** item of the **File** menu.

15.11 Exercises

In all exercises make liberal use of comments in your program files to clarify how the programs work.

1. Write a program for the PFW007, similar to that in Fig. 15−5, which calculates the sum of the integers between 1 and N, $\sum_{i=1}^{N} i$, where N is a number loaded into memory by the program file. The program should calculate the sum directly, by summing the integers, not by evaluating a formula such as Eq. (11-2). Run the program with the emulator and demonstrate that it works. Try the program with whatever values of N you like, but turn in a floppy with the program set up to run for $N = 6$ on it.

2. In this problem you are to write a program for the PFW007 that adds the absolute values of a table of numbers stored in memory. The size of the table (i.e. the number of numbers) is stored in memory location `0200H` (hex), and the table of numbers starts at `0202H`. Test your program using a table of at least five numbers, some of which are negative.

3. Write a subroutine program for the PFW007 that subtracts two numbers. For clarity lets call the two numbers n_0 and n_1. The program should calculate $n_0 - n_1$, and the numbers should be passed to the subroutine as arguments contained in `R0` for n_0, and `R1` for n_1. The result should be returned in `R2`. On return, the contents of all registers except `R2` should be left unchanged from what they were before the subroutine was called. Demonstrate that the program works by writing a main program which calls the subroutine, and run the whole thing through the emulator. Try your program with any numbers you like, but turn in a floppy with a program which subtracts 13 from 28 (both numbers expressed here in decimal).

4. Convert the subtraction subroutine in problem 15−3 so that arguments are passed to it on a stack. Demonstrate that it works the same way, and turn in a floppy with the same stuff on it.

5. Write a program for the PFW007 that writes a character string contained in memory to the output file. The end of the character string should be denoted by a zero, as in the program in Fig. 15−9. Demonstrate that it works by putting your name, capitalizing the first letter, in memory and writing it to the output file via the program. If your name has more than five letters in it, you can use just the first five.

6. Write a program which reads a character string from the input file and stores the characters in memory. The program should read in characters until it reaches the end of the file. Demonstrate that your program works by tacking on the program you wrote in the previous problem to write out what you've stored in memory to the output file. Test your program with the following politically correct character string.

   > To study, and when the occasion arises to put what one has learned into practice—
   > is that not deeply satisfying?
   >
   > Confucius, Analects 1.1.1

7. One very simple method for hiding the meaning of a message is to use the following idea to encode and decode the message. Each character of the message is encoded by adding a constant to the ASCII code corresponding to the character, and the character corresponding to the resulting ASCII code is transmitted. To decode the same procedure is followed, except that the constant is subtracted from each ASCII code. For example, if the constant were 16 (10 in hexadecimal) the character string "EE122" would be encoded to "UUABB".

   Write a program for the PFW007 which reads a character string from the input file, encodes it

using a constant stored at a specific location in memory, and writes the result to the output file. Write a second program which decodes the messages written by the first program. [HINT: This part's easy, just change the constant stored in memory.] Test your program using 13 (decimal) for the constant, and the following message.

> Everything should be as simple as it can be, yet no simpler.
>
> Albert Einstein

8. Write a program, similar to that in section 15−8, that reads a character string from the input file, assumes it to be a positive integer expressed in octal, and converts it into a number which could be stored in two bytes of memory. The program should assume that it has come to the end of the number in the file when it encounters a character which is not a legal octal digit.

9. Modify the program in the previous problem so that it can handle negative numbers as well as positive. You can assume that if the first character in the file is a `-` the number is negative, otherwise positive. Negative numbers should be stored following the two's complement convention.

10. Write a program which uses the `fact` subroutine in Fig. 15−19 to calculate the factorial of a positive number which is read from the input file. Do so by converting the program in Fig. 15−16 to a subroutine, and using it to read the number from the file. Just as in Section 15−8, the number in the file should be assumed to be the ASCII coded base 10 representation of the number. Test the program to convince yourself that it works. What value does it return for 7? For 8? For 9? Are these correct? If not, explain.

11. Write a program which reads a number from an input file, just as in the example in section 15−8, except use the approach first discussed in the third paragraph—push the ASCII codes for the individual digits onto the stack as they are read, and then convert the whole thing to the number by popping the digits off the stack, multiplying by the appropriate power of ten, and adding the results together. Test your program.

16. PROGRAMMING THE CPU WITH C

In this chapter, we'll start programming in C. I'll start with a brief introduction to C sufficient to allow us to write some simple programs, and then I'll conclude the chapter with a discussion of how to use the Turbo C compiler package to get programs compiled, run, and debugged. In the next chapter I'll continue the discussion of C. In both chapters I'll be emphasizing what the C statements actually cause the computer to do. As I think you will see, programming in C is much easier than programming in assembly language because the structure of a C program is much closer to the way people think than is the structure of an assembly language program, and because you don't have to worry about so many niggling details. Another advantage is that C hides many of the ugly details of the CPU. When programming in C, the compiler takes care of most of the unfortunate details associated with the particular CPU, and you need not even be aware that they exist.

This feature of C has an additional, pedagogical advantage. Statements in the C language tend to mirror general features of all CPU instruction sets. Thus what you learn in this course on a 80386-based machine will also be applicable on other computers using different CPU's with little or no modification. In several places, I'll draw upon the material we've already covered dealing with the fictional PFW007 to clarify what the C compiler is creating. I can do that even though you will be using C to program an 80386 CPU, because of the level of insulation from the instruction set details provided by C.

The main disadvantage of the approach is also pedagogical, and seems to be an unavoidable consequence of the generalized nature of C. The statements one uses in writing a C program are not very similar to the actual instructions the CPU will execute, and there is the concern that your understanding of the operation of the CPU may be more abstract than either you or I would like. That's the main reason I spent so much time on the "generic" instruction set of the PFW007, and the reason I'll try in this chapter to tie the C statements explicitly to how the CPU actually is programmed.

16.1 Elements of a C Program

In most programs, data will be stored in memory and retrieved from time to time by the program in order to operate on it. We encountered several such examples in the previous chapter. One of the advantages of programming in C (or in any high-level language) is that you need not worry about exactly where in memory the data will be stored. Instead, you can label each item of data with a unique name of your choice, and the compiler will take care of assigning an appropriate area of memory to each variable and remembering the address in memory where the data item starts. Such a labeled data item is called a *variable*. This is similar to my use of symbolic names for addresses in the assembly language versions of the programs in the previous chapter. The difference is that in the previous chapter, I was the "assembler", and I had to translate the symbolic names into actual addresses in the machine code; whereas here the C compiler will do all that for me. Besides relieving the programmer of a lot of tedium, this feature helps in designing and debugging the program. For example, suppose you were using C to do a calculation of voltages and currents in a circuit. You might have labeled the voltages in the circuit something like v_1, v_2, and v_3. Writing the program would be much easier, if the sections of memory containing each of these quantities were accessed using `v1`, `v2`, and `v3`, rather than addresses which might be something like `3C26A2`, `3C26A6`, and `3C26AA`.

In order to use variables, you have to tell the C compiler what variables you want to use, what type of data will be stored in each, and how much memory each will require. The C language statement that accomplishes this purpose is called a *declaration*. For example, suppose we want to use two variables named `a` and `b`, which are to be treated as integers. We need to tell that to the compiler, and would do so with the statement

```
int a, b;
```

Let's look in detail at this statement, first at the syntax of it, and then at what it accomplishes. The term `int` tells the compiler that this is a *declaration* statement, and that the variables being declared in the statement are to be considered as integers. The term `a` tells the compiler that one variable is to be called `a`, and the term `b` that another is to be called `b`. The two names are separated by a comma so that the compiler knows that we are declaring two variables rather than one with the name `ab`. Finally, the statement is terminated with a semicolon. In C, all statements are terminated thus. In many languages, like FORTRAN and BASIC, a statement is assumed to be ended when the end of the line is reached. If you want to spread a statement out over two or more lines, you have to tell the compiler that explicitly. In C, the situation is reversed. The compiler assumes that the statement continues until it reaches a semicolon, even if it spans two or more lines. Thus, we could have also written

```
int
  a,
       b;
```

This is pretty confusing to a human, but the C compiler would have no trouble with it, and would interpret it in exactly the same way as if it had been all put on one line.

I have been careless about spaces between the terms. C groups several characters into a set called *white-space* characters. Three of these characters are the space, the tab, and the new line characters. It is necessary that there be at least one white-space character in some places in C statements to separate items (like between the terms `int` and `a`) in order that the compiler know you intended to write two terms rather than just one, (`int a`, not `inta`). Other than that the compiler pretty much ignores white-space characters. In order to improve the readability of the code, you can put as many of them as you want almost any place where one can be used.

One final syntactical point: case makes a difference. Unlike many languages, C distinguishes between upper and lower case letters. Thus we could have two distinct variables, one named `a`, and the other named `A`. Also, the compiler will not know what you are talking about if you use `INT` or `Int` in place of `int` in a declaration statement.

Enough syntax, what does the declaration statement do? The statement does not cause any machine language code to be generated. Rather, it tells the compiler to set aside enough space in memory to hold each variable declared, and to associate each name with whatever information is stored in the corresponding space. Setting aside memory space for a variable is termed *allocation* of memory. Finally, the declaration statement tells the compiler how to treat each declared variable in subsequent operations involving it. For example, the declaration `int a` associates enough memory space to hold an integer, and associates the name `a` with the contents of this space. In subsequent statements involving `a` the computer will treat the number associated with `a` as a signed integer.

What exactly does the C compiler understand the term *integer* to mean? First, it assumes that the contents of the memory location associated with an integer variable represents a signed integer in binary. The number of bytes it uses to store an integer depends on the CPU. An integer is supposed to have the "natural" length for the CPU the program is to run on. For example, the Intel 8086 and 80286 are 16-bit machines, so an integer on a computer using one of these processors would be 16 bits long. The 80386 is a 32-bit machine, so an integer ought to be 32 bits long on it, but the compatibility problem appears again. Most C compilers for IBM PC's and clones make code that runs on the all the Intel 8086 family, so even on an 80386 an integer is taken to be 16 bits long.

If it were important that 32 bits be allocated to `a` you would use the specifier `long` in the declarative statement. Both

```
long int a;
```

and

```
long a;
```

are equivalent, and allocate four bytes of memory for `a`. Additionally, when doing arithmetic with `a` the compiler will be careful to keep at least 32 bits of accuracy, rather than 16. If it were important that only 16 bits be assigned to `a`, even on machines with 32-bit integers, you would use the specifier `short`. Both

```
short int a;
```

and

```
short a;
```

are equivalent, and ensure that only two bytes of memory will be allocated to `a`. Finally, if you wanted to allocate only one byte to a variable, you would declare it to be a `char`. For example

```
char a;
```

declares `a` to be a `char` variable using only one byte of memory. As you can probably guess from the name, `char`'s are used mostly to store character information. I'll discuss character strings in the next chapter.

All integers so far are assumed to be signed numbers, stored with the first bit being the sign bit and the number represented in the remaining bits. Negative numbers are stored as the 2's complement. For a 16-bit integer, that means integers between roughly −32,000 and +32,000 can be handled. C also allows you to store integers as unsigned numbers in which the first bit is not assumed to be a sign bit, but rather just one more bit in the number. In this case, integers between 0 and a little more than 64,000 can be accommodated. To allocate a variable named `a` as an unsigned integer, use the declaration

```
unsigned int a;
```

You can also use the `unsigned` attribute with `long` and `short`.

I want to discuss two more, related, variable types, `float`, and `double`. A variable declared to be a `float` is assumed to be stored with the single-precision, floating point convention. Recall from last semester that this convention is used to store fractions and numbers larger than the largest possible integer. A number is stored in three parts, a sign bit, a fraction, and an exponent. The number is the fraction times two raised to the power stored as the exponent. With the Turbo C compiler, a `float` occupies 32 bits or 4 bytes of memory, and numbers with magnitude between about 10^{-38} and 10^{+38} can be stored. The format in memory of the same number is quite different if it is stored as an `int` than if it is stored as a `float`. The `double` type is just a double-precision floating point number. This is a similar representation to single precision, except that 64 bits, or 8 bytes are used. You can also use the term `long float` for a `double`.

So far, the C statements we've encountered do not cause any machine code to be generated—they just tell the compiler what variables we want to use, and how we want them treated. I'd now like to look at a short set of statements that would actually cause machine language instructions to be generated. The program is pretty useless, but it gets us started. The program is the C language version of the PFW007 program in Fig.15−1 that subtracts 3 from 5. The program has very similar structure to that of the PFW007 program. I use three integer variables, named `a`, `b`, and `c`, storing the number `5` in `a`, `3` in `b`, and the difference of the two in `c`. Once the value `3` is stored in `b`, I negate it and add the result to `a`, just as in the program in Fig. 15−1. The C version of the program is shown in Fig. 16−1.

```
int a, b, c;

a = 5;
b = 3;
b = -b;
c = a + b;
```

Figure 16−1. C language program equivalent to the PFW007 program in Fig. 15−1 which subtracts 3 from 5.

The equals operator, `=`, in C is called the assignment operator, and it causes the value of whatever is on its right to be stored in the memory space associated with the variable on its left. The effect of the operator is similar to that of a `MOV` instruction. Thus the statement `a = 5;` causes the number `5` to be stored in the memory space associated with `a`, and corresponds to the second statement in Fig. 15−1. Most C compilers for Intel 80XXX microprocessors, including the Turbo C compiler on the computers in the lab, store `int`'s as 16-bit, signed, binary numbers. The next statement puts `3` in the space reserved for `b`. The next statement changes the sign of `b`, and corresponds to the third and fourth instructions in Fig. 15−1. The final statement causes the contents of `b` to be added to `a` using signed, 16-bit arithmetic, and the result to be stored in the memory space associated with `c`. Notice how much easier it is to deal with the C version of the program than with the assembly or especially machine language version.

Just to make sure I don't mislead you, there are better ways of writing the C program than above—I wrote it that way so that it would correspond to the PFW007 version. Better would be either to set `b = -3;` and then write `c = a + b;`, or to set `b = 3;` and then write `c = a - b;`.

Also, notice that the assignment operator is in some ways very different from the equals sign in algebra. The combination of instructions in Fig. 16−1 in which `b` is first set equal to `3` and then set equal to minus itself would be nonsense in algebra. Writing such a sequence of statements in algebra would require both conditions to be true *simultaneously*, and +3 is simply not equal to −3, period. In C, the equals sign means something different than in algebra. It means assign a value to the variable on the left side which is the same as the value of whatever is on the right. In C, the two statement sequence is perfectly sensible, and results in the value associated with `b` being `-3`.

In order to submit the program to the C compiler, the C program would be put into a file using any convenient editor, and the C compiler program would be run. As written above, however, the program would not compile. We need to add one little detail—we have to tell the compiler where the program module starts and where it ends. That may sound dumb at this point, because it's pretty obvious it starts at the start, and ends at the end. As you will see later, there are statements that can be placed before the start or after the end of the main program module, so in general it is not be so obvious what's prologue and epilogue. Further, the stuff that can be placed outside the main program can contain one or more subroutines, like the PFW007 multiplication subroutine in Fig. 15−9 for example, which the main program may call. It is, therefore, also necessary to tell the compiler which module is the main one. Curly brackets, `{` and `}`, are used to tell the compiler what group of statements are to be taken together as a module, and the term `main()` is used to tell it that the following module contains the main program, rather than some sub-program. Thus, the complete program would be

```
main()
{
    int a, b, c;

    a = 5;
    b = 3;
    b = -b;
    c = a + b;
}
```

Figure 16−2. A complete program in C which subtracts 3 from 5.

I hope the stuff in the preceding paragraph about the motivation for the curly brackets and the `main()` statement makes some sense to you, but you shouldn't worry too much if it doesn't. Just remember to enclose the main program in curly brackets, and put the `main()` statement before it. Consider it for now as part of a mystical rite if you like.

This is a complete program written in C. It's pretty useless, but it's legal. Notice the use of white-space (spaces, tabs, and new lines) to make it legible to humans. The only white-space that's required is the white-space between `int` and `a`. The rest is just to make it easier for humans to read and understand. As far as the compiler is concerned, the following is just as good. (Actually it's slightly better because it contains fewer characters that have to be processed.)

```
main(){int a,b,c;a=5;b=3;b=-b;c=a+b;}
```

Being of the human rather than computer persuasion, I find the first version much preferable. Since I, rather than my computer, am the one who writes the programs, I get to choose the writing style and all my programs look more-or-less like the first version. On the rare occasion when my computer writes a program, it gets back at me by concocting something that's even worse than the second version above. The style in Fig. 16−2 is more-or-less standard, and it works pretty well. It's not required, but I suggest you adopt a similar style.

16.2 Getting Results Printed Out

As I mentioned, the program in Fig. 16−2 is legal but useless. It's useless for two reasons. First, I already know what 5 − 3 is so this is a lot of trouble to find out something I already know; and second, I don't even get to find it out! The computer stores the result in the memory location associated with the variable named `c`, but it doesn't tell me what it is. I suppose I could run the program and then get into the circuitry of the computer with a voltmeter and a set of schematic diagrams and try to read the voltages associated with each bit in the two memory cells associated with `c`, but Sheesh!

What is needed is some way to make the computer print out the values of variables. To do that directly would not be easy. To print out a number on the monitor you would have to know many details about the interface circuitry used by the computer for the display, and write a fairly long and detail-filled program to send the correct data to the correct registers in the interface. You would also have to include in your program a section which converts a number stored in the format the computer uses into an ASCII string of characters in a format convenient for humans. Fortunately, the DOS operating system comes with some features called *system calls* which hide many of the details of communicating with the monitor interface circuitry, and C comes with a subroutine which hides the ugly details of converting from computer to people format for the numbers. The subroutine is called `printf`, and we can use it to print out all the variables in our program by adding a line as follows.

```
main()
{
    int a, b, c;

    a = 5;
    b = 3;
    b = -b;
    c = a + b;
    printf("a = %d   b = %d   c = %d\n", a, b, c);
}
```

Figure 16−3. Modified version of the program in Fig. 16−2 which prints the values of the three variables.

Since the `printf` subroutine is practically indispensable, I'll now discuss it in some detail. To discuss parts of it adequately we really need to have covered more material than we have but you will need `printf` to do anything useful with C, so I'll try to give you enough information here to use it. Subroutines are called *functions* in C, so the `printf` subroutine is properly called a function. Someone has already written the `printf` function, and the machine language for it (along with a number of other functions) resides in a file on the hard disk called a *library*. When the C compiler encounters the line in Fig. 16−3 containing the reference to `printf` it generates machine code to save the contents of all the registers, push the information referred to by the stuff between the parentheses onto the stack where `printf` can access it, and then jump to the first instruction in `printf` using an machine language instruction similar to the PFW007 `JSR`. The `printf` function ends with a machine language instruction similar to the PFW007 `RTS` (return from subroutine) instruction, so that when `printf` finishes execution of the main program continues with the statement following the `printf` statement. In this case, that's the end of the program but in general it will be another statement. This whole procedure is referred to as a *function call*, and we say that the main program *called* `printf`.

Information is passed to a function through its *arguments*. In C the arguments are placed inside a set of parenthesis and separated from each other with commas. For example, the call to `printf` in Fig. 16−3 passes `printf` four arguments. The second, third, and fourth arguments are the values of the three variables `a`, `b`, and `c`. The computer simply pushes the values of each of these variables onto the stack before jumping to `printf`. When `printf` needs these values, it simply retrieves them from the stack.

The first argument is a little more complicated. It contains an object we haven't discussed yet in this chapter, but we encountered in the last chapter, called a *character string*. The C compiler identifies a character string by the opening and ending quotation marks. The character string is made up of all the characters between the quotes, plus a terminating zero byte, and is stored sequentially in an array in memory with each character occupying one byte. The C compiler converts every character between the quotation marks into its ASCII equivalent, allocates enough memory space to hold the entire array, and loads the memory space with the ASCII equivalents of the characters in the string. As we discussed in the last chapter, the compiler places a zero in the byte immediately following the last character in the string to mark the end of the string. Before jumping to the `printf` subroutine, the computer pushes the address of this array onto the stack. Notice that it pushes just the address, rather than the whole array.

The character string in the first argument of the `printf` function call is called a *format*, and contains information about how the remaining arguments are to be printed. Almost all characters in the format string are printed out just as they appear in the string. Two exceptions to this statement are associated with the percent sign and the backslash characters, `%` and `\`. The percent sign and the characters immediately following it are used to tell `printf` to print out the value of an argument and how to print it out. For example, the first percent sign and the character following in the format string for `printf` in Fig. 16−3 indicates how the value of the first argument after the format is to be printed, the second percent sign and character following how the value of the next argument is to be printed, and so forth.

There are lots of possibilities for the stuff following the percent sign, and I'll only cover a few here. A lower case `d` indicates that the argument is a signed integer, and is to be printed base ten (i.e. in `d`ecimal). Instead of a `d`, we could have used a `x` or `X`, which would have resulted in the integer being printed out in base 16 or hexadecimal. In this case the corresponding argument is treated as an unsigned integer. The hexadecimal digits include the letters A-F. Using an upper case `X` causes `printf` to print these letters as capitals, and using a lower case `x` causes them to be printed in lower case. The letter `o` results in printing the corresponding argument in base 8 or octal. Again, the argument is assumed to be unsigned. You can use `g` to print out a `float` variable. For long variables, the letter should be preceded by a `l`. For example, for a `long int` printed in decimal, use `ld`, and for a `double` use `lg`.

At the end of the format string in Fig. 16−3 are the two characters `\n`. These two characters translate to the new line character, and they cause a skip to the start of the next line. This convention of a backslash followed by a letter is commonly used in C to refer to non-printing and special characters like new line. Table 16−1 lists the aliases for some such characters.

ALIASES

Alias	Non-Printing Character	ASCII Code (Hex)
`\\n`	New line	`0A`
`\\r`	Carriage return	`0D`
`\\t`	Horizontal tab	`09`
`\\b`	Backspace	`08`
`\\f`	Formfeed	`0C`
`\\\\`	Backslash	`5C`
`\\'`	Single quotation mark	`27`
`\\"`	Double quotation mark	`22`

TABLE 16−1. Aliases that can be used for some common, special characters

You may be wondering why special characters are needed for the backslash and the quotation mark since they both seem normal enough. Suppose you wanted `printf` to print out a quotation mark. You would then want to put a `"` inside the format string, but the compiler would get confused if you did because `"` characters are also used to denote the start and end of a character string. How is the compiler to know that you intended that one quote mark be used to produce a quote mark in the output, and the other two to delimit the character string? That's why the special meanings for sequences like `\"` are needed.

C has several characters that can have two meanings (like `"`). You can tell the compiler you want the secondary meaning by placing a backslash before the character. Thus the two-character sequence `\"` is interpreted by the compiler as an ordinary double quotation mark, rather than the start or end of a character string. Notice that the aliases listed in Table 16−1 for the non-printing characters follow this convention as well. For example `n` can have two interpretations. The primary interpretation is just the character n itself, and the secondary is the new line character. You get the secondary interpretation by prepending a backslash.

Anyway, the format string in the `printf` statement in Fig. 16−3 causes the computer to print out the following line.

```
a = 5   b = -3   c = 2
```

Just to illustrate some of the other formats, let's change the program so that `a` is set to −3, and `b` to 12, and I'll change the format string to print `a` in hex using lower case letters, `b` in hex using upper case letters, and

c in octal. The program then is

```
main()
{
    int a, b, c;

    a = 5;
    b = 3;
    b = -b;
    c = a + b;
    printf("a = %d   b = %d   c = %d\n", a, b, c);
    a = -3;
    b = 12
    c = a + b;
    printf("New values are:\n");
    printf("  In decimal:\n    a = %d   b = %d   c = %d\n", a, b, c);
    printf("\n  In hexadecimal and octal:\n    a = %x  b = %X  c = %o\n",
        a, b, c);
}
```

Figure 16−4. Modified version of the program in Fig. 16−3 which prints the values of the three variables.

Running this program results in the following lines.

```
a = 5   b = -3   c = 2
New values are:
  In decimal:
    a = -3  b = 12  c = 9

  In hexadecimal and octal:
    a = fffd   b = C   c = 11
```

I'd like to make a few points about the program in Fig. 16−4. First, to reiterate a point I made earlier, note that we first set a equal to 5, and then to −3. If we were doing algebra, that would not make sense, *a* may be either 5 or −3, but it cannot be both! As part of a C program, such a statement sequence is perfectly OK. In C the equals sign is more like the assembly language MOV instruction than the algebraic equals sign. In C, it just means place the value associated with whatever is on the right side into the area of memory set aside for the variable on the left. Thus, the first a assignment statement, `a = 5;`, stores the integer 5 in the two bytes set aside for a, and the second, `a = -3;` stores the integer −3 there, overwriting and destroying whatever was already there.

Notice also that there are two statements which add the values of a and b and assign the result to c. The first such statement stores in the two bytes associated with c the value of `a + b`, using the values of a and b *at the time the statement is executed*. These values are 5 and −3, respectively. Thus, the value 2 is stored in the memory associated with c after the first of these statements is executed. The same comments apply to the second occurrence of this statement, but this time the values of a and b are now −3 and 12, so after the second occurrence is executed the number 9 is stored in c's memory space. Similar comments apply to the two `printf` statements. Each time `printf` is called, the values of the arguments *at the time it is called* are passed to the `printf` function. The first time `printf` is called, these values are the character string with the first format stored in it, and 5, −3, and 2. The third and fourth times, `printf` is given different formats, and −3, 12, and 9.

Notice the use of the newline character, `\n`. If the new line character had not been in the first format string, the first call to `printf` would have caused the computer to print out the values of the variables and stop printing just after finishing the last one, without skipping to the next line. The second call to `printf`

would then start printing again at this point, and the output line would be

```
a = 5   b = -3   c = 2New values are:
```

Forgetting to put in new line characters everywhere they are needed in format character strings is a mistake that beginning C programmers make frequently and some experienced C programmers (like myself) make more often than they should. Fortunately, the error is easy to recognize and fix.

Notice that the second call to `printf` has only one argument, a format string. That's OK. There are no percent signs in the format, so `printf` doesn't look for any more arguments. Notice also the use of newline characters within some format strings to get the desired line spacing. Finally, notice that to improve readability I broke the last `printf` call into two lines. The line was getting too long to fit on the screen, so I simply continued it on the next line, and indented it a few extra spaces. I could have broken the line several places, but breaking it inside the format string would have introduced an unwanted newline character.

16.3 Compiling and Running C Programs

Once you have written a C program you need to translate it into the machine language your computer understands if you want to run it. In principle, if you spent enough time with the manual for the Intel 80386 you could do that by hand, but it would be a tedious, error-prone procedure. Fortunately, there exist special purpose programs called *compilers* which will do the translation for you. One such compiler is the Borland Turbo C compiler, and it along with several other auxiliary programs is available on the computers in the lab. The purpose of this section is to give you enough information about how to use the compiler and the auxiliary programs for you to be able to write and test your own C programs.

To use a C compiler, you first must put your C program into a file. For that you will need to use an editor program. Editors are programs which have a number of special features that make it easy for you to enter text into a file. You have already had to use an editor to put the programs for the PFW007 into a file. You can use whatever editor program you like best for the purpose, but there are some advantages to using the program supplied with the Turbo C compiler.

The main advantage of using this editor is that it is integrated with the compiler and several other programs supplied in the Turbo C package. Borland calls the whole thing the IDE, for Integrated Development Environment. The idea is that you start this one mongo program, the IDE, running and then you can do almost anything you want from it. From within the IDE you can create and edit your program file, compile it, combine the result with any subroutines contained in standard libraries (this process is called *linking*), load the resulting module into some convenient place in the computer's memory, run the module, view the output of the program, and debug the program in the event of errors. The advantage of doing everything from within the package is that the programs in the package are conveniently linked together. For instance if the compiler has a problem with your C program because of an error, when it returns with the bad news it starts the editor back up, puts the cursor of the editor on the offending line in your program, and opens a second window containing a message as to what the error was.

16.3.1 Entering, Compiling, and Running a Program

To start up the IDE enter `tc` at the DOS prompt. I've already given you a handout on how to use the editor that comes with the package. As I mentioned in that handout, the program seems to have a memory of what the last user was doing, and tries to start up doing where he/she left off. For that reason, when you start up you'll probably have one or more windows on the screen that you didn't create. Press **F10** to get the menu bar at the top of the screen, then **W** for **Window**, then **C** for **Close** as many times as needed to get a clean screen. (**Alt-F3** should work as well.) To start your own file, press **F10**, followed by **F** for **File**, followed by **O** for **Open**. Then type in the name you want to give the file and press **Enter**. If the file you are creating is to contain a C program, the extension should be `.C` because the compiler uses the extension to tell what kind of a program is in the file. That should give you a blue window that occupies about half of the screen. You can toggle between a half screen and full screen window with the **F5** key.

Use the editor to enter the program in Fig. 16−3 into a file named something like `PROG1.C`. I suggest indenting the lines similarly to the way they are indented in the figure. You can use the **Tab** key for that. Notice that the editor remembers the indentation of your last line and automatically indents the next line the same amount. That's a feature specially tailored to writing C programs. If the editor indented too much use the backspace key.

After you have your program typed in, you can compile it by pressing **F10**, followed by **C** for **Compile**, followed by **M** for **Make EXE**. A window should pop up with a sign on top saying `Compiling` at the top. After a short time, that should change to `Linking`. I'll discuss what this means shortly. If you have typed the program into the computer without errors, `Success` should appear on a blue line at the bottom of the window. Pressing any key makes the window disappear. There should be a sort-of aqua colored new window at the bottom of your screen with the title `Message` at the top. This window gives you a report on what the compiler did. If you entered the program correctly, there should be three lines, the middle of which is a warning saying that a function should return a value. Don't worry about it, the problem is more one of etiquette than substance, and the program will run fine.

If you erred in typing in the program there will probably be a list of several errors in this window. I'll assume you got it typed in OK. If not, and you can't figure out how to fix it, look in subsection 16−3.3 to find out how to fix the mistakes, fix them, and then come back here. At this point, the active window will be the `Message` window, probably numbered `2` in its upper right hand corner. To do anything, you have to make the editor window the active window. In general, you can shift to another window by pressing **Alt** and the number of the window you want at the same time. For example, if the editor window is numbered `1`, you would press **Alt** and **1**. In this special case, you can also get back to the editor window by pressing **Enter**. Still a third alternative is to press **F10**, **W**, and **L** for **List**. That will give you a list of all your open windows, and you can go back to any one of them by selecting it on the list and hitting **Enter**.

Choose one of these methods and get back to the editor window. Assuming it compiled correctly, you can run the program by pressing **F10**, followed by **R** for **Run**, followed by **R** for **Run** (I did not just stammer). The screen should blink, and you'll be back to your program file. The blinking was the program writing out to the screen the values of `a`, `b`, and `c` as instructed in the `printf` statement. I'll bet you didn't catch what it wrote to the screen! To see what happened at a more leisurely pace, press **F10**, **W**, followed by **O** for **Output**. That should produce a window which shows the last screenful of whatever has been written to the screen. You won't see the entire screenful because the window isn't full size. If the program wrote to the bottom of the screen (and it probably did), the output will be below the bottom of the window. You can see it by scrolling with the arrow keys, or by pressing **F5** to get a full-size window.

16.3.2 What's Really Happening

Before continuing, I'd like to discuss briefly what is happening when you **Make EXE** from the **Compile** menu, and then run your program. When the pop-up window says `Compiling` at the top, the program is busy translating the C language code into machine language. At the end of this phase there may be some addresses still unknown, and these are marked in the file. The result is what is called an *object* file, identified by a file extension of `.OBJ`. This file contains the machine language translation of most of the C program and some tables for the use of the linker program. The file cannot be executed as is because some important stuff is missing. For example, in our program we call the function `printf`, but we do not provide the program code for `printf`. Someone has already written and compiled that subroutine and placed the object code for it along with a number of other generally useful subroutines in a file called a *library*.

When the pop-up window says `Linking` at the top, the computer is running another program called the *linker* or *link editor*. This program searches several standard library files trying to find any subroutines it needs but doesn't have. When it finds them, it tacks them onto the end of the machine language code for your program, and adjusts the addresses in your program accordingly. The linker also adds some standard code onto the start of your program so that when DOS transfers control to your program it will start up properly and return properly to DOS when finished. The linking phase is also sometimes referred to as *loading*. I think there is some fine distinction between the two terms, but it seems to have become pretty

well blurred. For this class, you can think of the two terms as being synonymous.

The result of all this is an *executable* file, identified by the `.EXE` extension. All the activity up to now is analogous to what you did by hand in preparing a program file for the PFW007, although the `.EXE` file is not meant to be read by humans, and has no comments in it, of course. When you run the C program out of the **Run** menu of the IDE, or by simply typing in the name of the executable file from the DOS prompt (you don't have to include the `.EXE` extension), DOS copies the machine code in the file to some part of memory, sets up the segmentation registers and other messy stuff, and then jumps to the first instruction of your program. Your program does its thing, and when finished, it jumps back to the DOS program (DOS and the linker set it up that way). The activity of the computer during this phase is similar to the activity of the PFW007 emulator when you ran it. The emulator read the machine language program from the file, copied it into its memory, executed it, and then returned to DOS.

16.3.3 Compile-Time Errors

Some errors you make in writing a C program will be caught by the compiler, and are called *compile-time* errors. Others won't be discovered until you try to run the program, and these are called *run-time* errors. In this subsection, I'll discuss the first kind, and in the next session the second. Compile-time errors are usually the result of some grammatical or syntactical indiscretion, such as omitting a semicolon or having more left parentheses than right. The compiler is quite helpful in tracking down such errors. To illustrate how it works, lets purposely insert an error into the simple program we've been discussing. Using the editor, delete the semicolon at the end of the `b = 3;` statement. Compiling the program now will produce a pop-up `Compiling` window with `Errors` at the bottom. That means the compiler has found an error which is serious enough that it can go no farther. Pressing any key activates a `Message` window with the second line high-lighted. It says `Statement missing ;`, and in the editor window the line after the offending line is high-lighted. The compiler is trying to tell you where it found the error, but it missed by one line.

To fix it, you first have to activate the editor window, and then insert a semicolon at the end of the line just before the cursor. After doing that, the program should compile and run normally. If you had multiple compile-time errors, you would then have made the `Message` window active again, moved the highlight to the next item, gone back to the editor window, fixed the problem, and so forth.

Don't take the error messages from the compiler too literally. The compiler is only a dumb program, and it doesn't always diagnose the problem correctly. If the compiler says something is wrong, it's almost certain something is wrong, but the error message may not accurately tell you what it is. The message usually does point you to the general area where there's a problem. In the example we just considered, the message correctly told us that there was a missing semicolon, but it put the cursor on the line following the problem, rather than on the offending line.

The interaction between the compiler and the editor is a result of the linkage between the two programs provided by the IDE package. You could also use your own editor and run the compiler independently. In that case the compiler would report the errors it finds to the screen, giving you the line number where it thinks the error is, and a brief description. You would have to either remember or write down all this information, start up your editor, move to the offending line(s) and make the correction(s), exit the editor, and then run the compiler again. Life with the integrated package is easier.

There is also a debugging program provided with the package that lets you run your program line-by-line, checking the values of variables as you do. In the IDE package this program is even more closely integrated with the editor and compiler. I'll discuss the use of this program in the next subsection.

16.3.4 Run-Time Errors

Run-time errors are often quite a bit harder to find than are compile-time errors. There are two general types of run-time errors. In the first, some error is made that causes the CPU to encounter an illegal instruction, or to access some non-existent part of memory. Usually in such cases, the computer simply stops executing the program and returns you to DOS with a haughty message, but occasionally the faulty

program may misbehave so badly that it messes up the DOS program. Then the computer will go catatonic, and you'll have to reboot. The second type is an error for which the program runs apparently normally, but with incorrect results. These errors can be the worst to find.

The Turbo C package comes with a debugging facility which can be helpful in tracking down these kinds of errors. The debugger has several features, but I'll touch on only a few here. To illustrate, I'll introduce a run-time error into the simple program we've been discussing. This program is so simple, that it's hard to put in an error that you'd need the debugger to find, but I'll profess to be even less observant than I am so that I can show you how the debugger works.

I'd like to consider two such errors. For the first, I'll change the line that should read `b = -b;` to read `b = +b;`. That will be my simulated error—I mistyped and entered a `+` instead of `-`. Inserting that mistake, the program compiles without error and runs apparently correctly, except that it prints out `c = 8` instead of `2`. From the printout it's pretty obvious what's wrong, because the program also prints out `b = 3`, not `-3` as expected. I'll pretend I didn't notice this, though, and use the debugger to help figure out what the problem is.

To do that, I'll use the debugging facility to execute my program, one line at a time, and I'll set what Turbo C calls *watches* on each of the variables, `a`, `b`, and `c`. Setting a watch causes a new window to appear at the bottom of the screen, with the names of all the watched variables in it. As the program is executed, this window keeps track of the current values of each of these variables. Set a watch on `a` by pressing **F10**, then **D** to get the **Debug** menu, followed by **W** for **Watch**. A window will pop up asking you what you want to watch. Answer `a`. A new window should appear at the bottom of the screen with `Watch` at the top. The first line should read something like `a: Undefined symbol 'a'`. You haven't run the program yet, and the line just means that the watcher doesn't know what the heck `a` is yet. Do the same thing to set watches on `b` and `c`. Your watch window now should have three lines in it, one for each of the variables.

Now we're ready to run the program. We don't want to run the whole thing at computer speed because it will happen too fast to follow. Instead, we will start the program executing, but have it pause just before executing the instructions associated with the first code-producing line in the C program. From that point on, we will execute one line at a time, pausing after each to see what happened. To get this started, make the editor window the active window and position the cursor on the first line of the program that generates code, the `a = 5;` line. Then press **F10** followed by **R**, followed by **G** for **Go to cursor**. The computer should run at its own speed until it gets to the line you marked with a cursor, and stop before it executes it. The screen will blink and soon there will be an aqua-colored high-light over the line your editor cursor was on, the one setting `a` to `5`. That means that the computer has gotten to this point in the program, but has not executed the machine instructions associated with this line yet. Looking in the watch window, the variables are no longer undefined, although they may have some strange values. The variables have been assigned memory space, but we haven't put anything in this space yet. The watcher is simply reporting whatever value was left there by the last user of the space.

Now press **F8** to *single-step* the computer. The computer will execute all the machine language instructions associated with the `a = 5;` line, and then stop before executing anything associated with the next line. That's what's meant by "single-step." Notice in the watch window that `a` now has the correct value. Press **F8** again, and the next line is executed. The high-light moves down one line, and `b` now has the correct value. Pressing **F8** one more time causes the faulty line to be executed. Afterwards, `b` still has the value `3`, instead of the expected `-3`. Aha! There's the problem!

We need to fix the corresponding line and recompile. To get out of the single-stepping stuff choose **Program reset** from the **Run** menu. Make the correction, recompile, and run the program. Note that if, after making the correction, you choose **Run** from the **Run** menu, the IDE is smart enough to realize that the C program has been changed, and needs to be recompiled. Before it runs the program, it compiles and links it.

I'd like to look at a more realistic run-time error that could be more difficult to find. In the last line of the program, delete the final argument to `printf` so that the line now reads

```
printf("a = %d  b = %d  c = %d\n", a, b);
```

This time my simulated error will be that I forgot to type in one argument to `printf`. The format string claims that there are three variables to be printed out, one for each of the `%d`'s, but only two are supplied. The `printf` function uses the format string to figure out how many arguments there are, and believes the string to be correct. It looks, therefore, on the stack where the third variable should have been put, and prints out whatever is there. Apparently, that number was `6`, because that's what the program printed out for the value of `c` when I ran it like this.

The compiler does not recognize my mistake as an error because the `printf` function is simply a subroutine that someone has written. The compiler has no way of knowing that the number of arguments it should have depends on the number of `%` terms in the format string. At run-time, no error is discovered because the place on the stack where `printf` looks for `c` is a legal address, and the computer has no way of knowing that the value contained therein is wrong. Finding the mistake could be troublesome because the error could either be something like the error we introduced previously, where the code is all legally correct but carries out the wrong calculation, or some illegality such as the incorrect number of arguments to `printf`. Which is it? Practically the whole program is suspect and it's hard to know where to look. The debugger would be helpful in this case because we could set a watch on `c` and find that `c` had the correct value just before calling `printf`, fingering something about the `printf` call pretty clearly.

In a more complicated program, there are a lot more variables, and setting watches on each of them is a problem. As an alternative, you can single-step through the program, and at any time find out the current values of any variable by using the **Evaluate/modify** item of the **Debug** menu. Another helpful feature for longer programs is that once the debugger is in the single-step mode, you can go rapidly through a set of instructions, like a loop for example, by moving the cursor to where you want to pause next, and choosing **Go to cursor** from the **Run** menu.

16.4 Exercises

1. The C program in Fig. 16−3 is not as similar to the corresponding program for the PFW007 in Fig. 15−1 as it could be. In the C program, the number called `b` is negated in one step, with the statement `b = −b;`. There was no corresponding single instruction for the PFW007 so changing the sign of the number in `R0` was done in two steps: `NOT`'ing the number, and then adding one to it. Rewrite the program in Fig. 16−3 to negate `b` in this way. The operator in C which performs the `NOT` operation is `~`. Thus, to `NOT b`, one would write `~b`. Compile and run your program to show that it works.

2. In the printout from the program in Fig. 16−4, the value contained in `a` the second time it's printed out is `−3`, but when the same variable is printed out in the next line in hex, the value is given as `fffd`. Explain.

3. Type the program in Fig. 16−5 into a file, compile it and run it.

```
main()
{
    int a, b, c;

    a = 31000;
    b = 8000;
    c = a + b;
    printf("a = %d  b = %d  c = %d\n", a, b, c);
}
```

Figure 16−5. Program for exercise 16−3

   Explain the output.

4. Type the program in Fig. 16−6 into a file and compile and run it.

```
main()
{
    float x, y, z;

    x = 1.e32;
    y = 1.e12;
    z = y*z;
    printf("x = %g  y = %g  z = %g\n", x, y, z);
}
```

Figure 16−6. Program for exercise 16−4.

   The term `1.e32` means 1×10^{32}. This notation is similar to that used in Quattro Pro. The program should complete abnormally (that's a polite way to say "bomb out"). Explain what happened. Set a watch on `x`, `y`, and `z` and single-step through the program, starting just before executing the `x = 1.e32;` statement. For each step describe and explain the contents of the Watch window.

5. The resistivity, ρ, of a semiconductor is given by the formula

$$\rho = \frac{1}{q_e(\mu_e n + \mu_h p)}$$

   where q_e is the charge on an electron (1.602×10^{-19} Coul.), n and p are the densities of free

electrons and holes in the semiconductor, and μ_e and μ_h are the mobilities of electrons and holes in the semiconductor, respectively. Write a C program which determines the resistivity for the following values of n, p, μ_e, and μ_h.

$$n = 2.6 \times 10^{14} \text{ cm}^{-3} \qquad p = 8.3 \times 10^{14} \text{ cm}^{-3}$$

$$\mu_e = 1450 \text{ cm}^2/\text{V} - \text{sec} \qquad \mu_h = 450 \text{ cm}^2/\text{V} - \text{sec}$$

Your program should print out the values of n and p along with the corresponding value of σ in the following format.

```
n = whatever (cm^-3)
p = whatever (cm^-3)
resistivity = whatever (Ohm-cm)
```

17. MORE PROGRAMMING THE CPU IN C

In this chapter, I'll continue the introduction to C started in the last chapter. Now that you know how to compile and run your own C programs, we will be able to go into greater detail. The emphasis here will be on using C to poke around in the computer to see how things are stored, and the like, but I'll also introduce a number of important features of C, and discuss how they can be used in a program.

17.1 Pointers

One of the advantages of C over many other high-level languages is that with C you can directly access and use the memory address of variables. For this purpose C has a class of variables called *pointers*. A pointer to an integer, for example, is a variable which is used to contain the memory address of an integer variable. There are as many types of pointers as there are types of variables. You can have a pointer to a `char`, a `long`, a `float`, a `double`, or even to another pointer. The discussion at this point can get a little abstract, so lets consider a specific example. Suppose we have declared the variable `a` to be an integer, and the compiler has assigned `a` the memory addresses `8000H` and `8001H`. Suppose also that `a` had been set equal to 12 earlier in the program (that means that the value 12 was written into these memory locations), and that we have declared `p` to be a pointer to an integer. (I'll show you how to make such a declaration shortly.) We could then set `p` equal to the address of `a` with the statement

```
p = &a;
```

The ampersand symbol used in this context is the "address of" operator, and the statement would cause the value of `p` to become `8000`, the address of `a`.

There is also a kind of inverse operator to the "address of" operator which you might call the "contents of" operator. If I had one integer variable, `b`, and I wanted to set it equal to another, `a`, I could just write `b = a;`, or I could write (after executing the code just above setting `p` equal to the address of `a`)

```
b = *p;
```

The "contents of" operator can also be used on the other side of the equals sign, as in the following statement in which the value of `a` is set equal to `5` with the following statement

```
*p = 5;
```

The asterisk is also used to indicate multiplication. You might think that this double use would cause confusion, but it doesn't seem to.

To tell the compiler that the variable `p` is to be a pointer to an integer, you would use the following declaration.

```
int *p;
```

This may seem like a strange way to make the declaration, but there is some method to the madness. It says that `*p` is an integer, implying that `p` must contain the address of an integer, and `p` must therefore be a pointer to an integer. The method has the advantage that it can be easily generalized to declaring a pointer to any kind of a variable. For instance, we could declare the variable `usp` to be a pointer to an unsigned short integer with

```
unsigned short *usp;
```

Pointers are frequently useful in writing programs, but at this point I can offer you only a few examples. One example involves communicating directly with an interface circuit for a peripheral device. When

you want your program to communicate directly with a peripheral device you must write data to or read data from specific addresses. For example, the PFW007 emulator was set up so that to write data to a file you had to `MOV` bytes to the special address `F000`. Real computers are set up similarly, although the interaction between peripheral and computer is often more complicated. With many high-level languages accessing a specific address is difficult or impossible, and for this reason these languages are seldom used to write such programs. With C it's easy. If we had a C compiler for the PFW007, and wanted to write the number 78, for example, to the output file, we would simply say

```
*(0xF000) = 78;
```

The `0x` before the `F000` simply tells the C compiler that this number is expressed in hexadecimal rather than decimal. If, on the other hand, we wanted to read a number from the input file, and put it in an integer variable named `a`, we would use

```
a = *(0xF000);
```

17.1.1 Rear Window I: Addresses of Variables

We can use pointers to help snoop around the private life of the C compiler. For example, we can use the "address of" operator to find out where in memory the C compiler places our variables. This is something you would not usually want to know, but this isn't a usual situation since here we're more interested in learning what's really going on inside the computer than in writing a useful program. I'll add a line to the five minus three program in Fig. 16−3 to print out the addresses of `a`, `b`, and `c`. The result is shown in Fig. 17−1.

```
/*
 *  Program to subtract 3 from 5.
 *  The program prints out both the results and the addresses
 *  used by the compiler to store the variables.
 */

main()
{
    int a, b, c;

    a = 5;
    b = 3;
    b = -b;       /* Negate b */
    c = a + b;    /* Add -b to a */

/* Print out the results */
    printf("a = %d   b = %d   c = %d\n", a, b, c);

/* Print out the addresses used */
    printf("Address of a = %x, of b = %x, and of c = %x\n", &a, &b, &c);

    return 0;    /* Return a value so the compiler stops griping at me */
}
```

Figure 17−1. Modified version of the program in Fig. 16−3 which prints the addresses of the three variables.

There's one complication here. If you try to compile any of the programs in this section, make sure the compiler is set to produce code for the "Tiny Model." To do that choose `Options` from the menu bar, followed by `Compiler`, followed by `Code generation`. That should give you a menu with a number

of items, some of which are in a column labeled `Model`. Choose `Tiny` from this column by using the arrow keys. If you are wondering what this is all about, it's related to the segmentation registers in the Intel microprocessors. I don't want to go into it here and it's not important for understanding the stuff in this chapter, but in case you are curious I discuss the unpleasantness briefly in the appendix to the chapter.

I've included a new feature in the program, comments. These are parts of a program which the compiler ignores, and which are included for the convenience of humans. Comments should be used to make your programs more readable and understandable, both by others and by yourself when you come back to it after some time. A comment is started by the two-character combination `/*`, and terminated by `*/`. It should be pretty clear where the comments are in the program, and how they are used.

I have also fixed up in the program the little breach of etiquette that the compiler keeps complaining about. The main program is something like a subroutine called from DOS, and it can return a value to DOS, very much like the subroutines we wrote for the PFW007 returned a value to the calling program. DOS is too stupid to be able to do much with it this value, but C can be used with other operating systems such as UNIX which can. The program in Fig. 17−1 did not tell the compiler what value is should return when `main()` is finished, and this is what the compiler is complaining about. There are two ways to placate it. The way I chose here was to add the instruction `return 0;` at the end of the program. This tells the compiler to return the value 0 (it could just as well been any other value) to DOS when the program is done. The other way would have been to change the first non-comment line to read `void main()`. The `void` tells the compiler that `main()` is not supposed to return any value. In this case, it would be an error to include a `return` statement in `main()`. I will discuss these features in more detail in the next chapter.

Compiling and running this program causes the following two lines to be printed to the screen.

```
a = 5  b = -3  c = 2
Address of a = fff4, of b = fff2, and of c = fff0
```

The variable addresses start at nearly the top of the address space available (remember the Turbo C compiler programs the 80386 as if it were a 16-bit machine) and decrease in the order they were declared. The reason for this is that C treats the variables as automatic variables and gets space for them from the stack.[3]

As an aside, we could also have put the declaration for the variables outside the main program as in the listing in Fig. 17−2. In this case the printout is

```
a = 5  b = -3  c = 2
Address of a = 170a, of b = 170c, and of c = 170e
```

The C compiler considers variables declared outside the main program, and outside any subroutine to be *global* and to be available for use by any subroutines occurring in the program file after the declaration. For this reason space for the variables is not obtained from the stack, but rather permanent space just after the program is assigned for storage of the variables.

17.1.2 Rear Window II: Byte Ordering

We can also look at how the compiler stores numbers in memory. An integer is stored in two bytes of memory, and the question is in which order are the bytes placed. Is the CPU little-endian or big-endian? I wrote the program shown in Fig. 17−3 to answer the question. An integer variable, `a`, is declared and set

3. Actually, the addresses reported here are not quite the actual addresses where the data are stored because the addresses reported must be combined with the appropriate segmentation registers to produce the actual address. Turbo C has the capability of accessing these registers, so you could get that information too if you wanted, but this is one can of worms I suggest you avoid if at all possible.

```
/*
 *  Program to subtract 3 from 5.
 *  The program prints out both the results and the addresses used
 *  by the compiler to store the variables.  In this program,
 *  the variables are declared externally, rather than internally.
 */

int a, b, c;    /* The external declaration */

main()
{
    a = 5;
    b = 3;
    b = -b;       /* Negate b */
    c = a + b;    /* Add -b to a */

/* Print out the results */
    printf("a = %d   b = %d   c = %d\n", a, b, c);

/* Print out the addresses used */
    printf("Address of a = %x, of b = %x, and of c = %x\n", &a, &b, &c);

    return 0;    /* Return a value so the compiler stops griping at me */
}
```

Figure 17−2. Modified version of the program in Fig. 16−3 which prints the addresses of the three variables. In this program, the variables are declared outside the main program.

equal to `1234H`. To answer the question of the day, we print out the values of the individual bytes of `a`. Doing that is a little tricky. I declare a variable, `p`, to be a pointer to an `unsigned char`, and set it equal to `&a`. Then `p` will contain the address of the start of `a` in memory, and `*p` will refer to the byte at the lower address. Similarly, `*(p+1)` will refer to the byte at the next higher address. There is a minor problem: `printf` does not have a format item to print out a `char` as an unsigned integer in hex, so I declared two integer variables, `lobyte` and `hibyte`, and set `lobyte = *p` and `hibyte = *(p+1)`. That lets me use the `%x` format item to print out `lobyte` and `hibyte`.

Compiling the program gave a warning message that I'll discuss shortly. It can be ignored for now. The result of running the program is

```
The two bytes of a are:  34, 12
```

As you can see, the compiler stores the least significant byte in the lowest memory location, implying that the computer uses the little-endian convention. Just to check, I ran the same program on another computer which uses the Motorola 68010 CPU. This CPU uses four byte integers, so I had to modify the program slightly to print out four bytes rather than two. The result was:

```
The four bytes of a are:  0, 0, 12, 34
```

As you can see, the 68010 CPU uses the opposite byte ordering convention to that used by the Intel 80386. The 68010 is big-endian.

I want to make a few comments about the program. First, the statement `lobyte = *p;` causes the compiler to make a conversion. The quantity on the right, `*p`, is an `unsigned char`, and the quantity on the left, `lobyte`, is an `int`. The internal formats of these two variables in memory are a little different; `*p` occupies one byte and `tmp1` occupies two. When the compiler sees the assignment operator connecting these two variables, it automatically generates code to convert the quantity on the right into the format of the quantity on the left, and stores the converted value in the memory space of the left-hand variable.

```
/*
 * Program to investigate the byte ordering of integers.
 */

main()
{
    int a, lobyte, hibyte;
    unsigned char *p;

    a = 0x1234;

/* Set p to point at the first byte of a */
    p = &a;

/* Put the two bytes of a in lobyte and hibyte */
    lobyte = *p;
    hibyte = *(p+1);

/* Print them out */
    printf("The two bytes of a are:  %x, %x\n", lobyte, hibyte);

    return 0;
}
```

Figure 17−3. Program to print out the values of the two bytes of an integer in the order they are stored in memory.

This is usually what you want it to do. It was in our case because we wanted the value of `lobyte` to be the same as the value of `*p`, even though the internal formats are different.

The conversion from an `unsigned char` to an `int` is pretty straight-forward. The compiler just places the contents of the byte associated with `*p` into the appropriate byte of `lobyte` (for a little-endian machine that's the byte having the lower address). Conversions are not always so simple, though. Consider what would have happened if we had declared `p` to be a pointer to a `char`, rather than an `unsigned char`. Then `*p` would be interpreted as a *signed*, one-byte number. Consider what would have happened if `a` had been set equal to `-3`, for example, instead of a `1234H`. The 16-bit version of `-3` is `fffdH`, and if `p` were set to point to the least-significant byte of `a`, `fd`, it would be interpreted as a signed eight-bit number, −3. The assignment `lobyte = *p` would result in the value of `lobyte` being set to −3. Since `lobyte` is an `int` it occupies 16 bits rather than 8, and −3 is stored in an `int` as `fffd`, not `00fd`. The `printf` statement would then print out `fffd` for `lobyte`, not `fd` as desired. Similarly, it would print out `ffff` for `hibyte`.

I'd like now to discuss the warning message the compiler gave when compiling the program. The message was concerned with a "Suspicious pointer conversion," and was triggered by the statement setting `p` equal to the address of `a`. The variable `p` is supposed to point to an `unsigned char`, but the assignment statement tells it to point to an `int`. The C compiler therefore does a conversion, converting the address of an `int` to that of an `unsigned char` before performing the assignment. The address of an `int` is stored in exactly the same format as that of an `unsigned char` so the conversion is trivial—the value of `p` is simply set equal to the address of the `int`. The reason the compiler issued the warning was not that there was a problem with the conversion, but rather that it recognized a potential problem with the *portability* of our program to other computers. On a computer which uses a different byte ordering for integers, our program would give different results. This happens to be just what we want it to do in this special case because the program is supposed to determine the byte ordering used by the computer, but usually you would not want the same program to run differently on one computer than on another. This statement would then be a rather difficult to find bug.

17.2 Arrays

You can reserve a block of memory for an array of variables of any type. The most common use of this feature is probably the storage of character strings, as we discussed in the previous chapter on the PFW007. Recall that textual information is stored in the computer as a sequential list of one-byte numbers. In C, you would set aside memory for storing a character string by declaring an array of `char`'s. For example, to store the character string "Hello", we would require five bytes of memory for the five characters in the string, plus an additional byte for the terminating zero. We would allocate the memory needed for storing "Hello", and give it a name, say `greeting`, with the following declaration.

```
char greeting[6];
```

This declaration sets aside six bytes of memory as an array of `char`'s. The variable `greeting` by itself is a pointer to a `char` which the compiler sets equal to the starting address of this block of six bytes, the first `char` in the array. We can access, say, the third byte of this array with the expression `greeting[2]`. The `2` is correct; counting starts with `0`. We've allocated the memory for the array, but we have not yet put anything in it. To put our character string in it, there are several ways to proceed. One way would be to do it by hand:

```
greeting[0] = 0x48;    /* H */
greeting[1] = 0x65;    /* e */
greeting[2] = 0x6c;    /* l */
greeting[3] = 0x6c;    /* l */
greeting[4] = 0x6f;    /* o */
greeting[5] = 0;       /* End of string */
```

Instead of looking up the ASCII equivalents for the characters, we could let the C compiler do it for us:

```
greeting[0] = 'H';
greeting[1] = 'e';
greeting[2] = 'l';
greeting[3] = 'l';
greeting[4] = 'o';
greeting[5] = 0;
```

The C compiler recognizes a character between single quotes to mean that you want the ASCII equivalent code of the quoted character. (Well, not always ASCII. Some computers still use other character codes such as EBCDIC, and a C compiler for such a computer would translate to the native character code instead of ASCII.)

There are other ways of copying the character string into the `greeting` array. The best way I know of uses pointers and a loop. We haven't discussed program control statements in C yet, so I can't talk about that now and I'll defer the discussion to a later section. Another approach to the problem is a kind of an end run. Instead of declaring `greeting` to be an array of `char`'s, we could declare it to be a pointer to a `char`, and write the following code.

```
char *greeting;

greeting = "Hello";
```

The first line is simple, it just declares `greeting` to be a pointer to a `char`. It allocates two bytes to hold

the value of `greeting`, but no other memory is set aside. The second line requires some discussion. When the C compiler sees a string of characters between double quotes, as in the example above, it sets aside memory space for the string, and puts the ASCII equivalents for each character plus a terminating zero byte into the space. The value of a quoted string, like `"Hello"` above, is not the string, but the address of the start of the array holding the string. Thus the second line in the program fragment above sets `greeting` equal to the address of the start of the string in memory. Notice that it does *not* copy the string on the right to some space associated with the pointer on the left; it only sets the value of `greeting` to be the address of the string. If you will go back to section 16−2 where we discussed the `printf` function, you should see that the compiler treated the format string the same way. The value the compiler pushed onto the stack for the format string when it called `printf` was the address of the string, not the string itself.

There are some subtle differences between the two approaches, but the results are similar. With this approach, if we wanted to access the second character of the string, we could use the expression `greeting[1]`, just as if `greeting` were an array, or we could use `*(greeting+1)`.

Notice that the following code fragment would not work, and should cause the compiler to flag you for an illegal procedure penalty.

```
char greeting[6];

greeting = "Hello";
```

You might expect the second line to cause the characters of the string on the right to copied into the array pointed to by `greeting`, but that is not how the C compiler interprets the statement. The compiler interprets it as saying that `greeting` should be changed from pointing to the array allocated in the declaration statement to pointing to the start of the "Hello" character string. That's an error because the name `greeting` is irrevocably tied to the address of the start of the memory block allocated in the declaratory statement. The compiler interprets the second line as trying to change that assignment and balks. It's something like writing `3 = 2;`. The value of `3` is irrevocably 3, and trying to change that is illegal.

You can declare arrays of integers, or any other variable type using analogous syntax. For example, if you were writing a program to analyze the test scores of a class, you might want to declare an array of integers, maybe named `scores`, to hold the scores for analysis. If there were 40 students in the class you would use the following declaration

```
int scores[40];
```

A block of memory big enough to hold 40 integers (80 bytes with Turbo C) would be reserved, and the value of `scores` by itself would be the address of the start of this memory block. You could then put the test scores in this block and access them with expressions like `scores[12]` or `*(p+12)`. You could even declare an array of pointers to arrays of characters to contain the names, but we won't go into that here.

17.3 Conditionals, and Program Flow and Loops

In this subsection, I'd like to discuss how C handles conditional execution of code (analogous to the use of `BR` instructions with the PFW007), and program flow.

17.3.1 Conditionals

It frequently occurs in programming that you want a block of code executed only if some condition is met. In C the `if` and `if ... else` statements are used for this purpose. For example, suppose in a program you needed to calculate the absolute value of a number. You might use the following program fragment.

```
/*
 * A program fragment to set y equal to the absolute value of x.
 */
    float x, y;

    if(x < 0.)
        y = -x;
    else
        y = x;
```

I'd like to look closer at what's inside the parentheses in the `if` statement above. In C `if` must be followed by an expression enclosed in matched parentheses. If the expression is non-zero, the statement following it will be executed; otherwise it won't. The "object" `x < 0.` is an expression with a value. The value of the expression is `1` if the relationship is true, `0` if it's false. The expression is called a *relational* expression, and the `<` operator is called a *relational* operator.

C has a number of such relational operators, and I list them in Table 17−1.

RELATIONAL OPERATORS

Operator	Meaning
==	Equal
!=	Not equal
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

TABLE 17−1. Table of relational operators in C.

These are pretty much self-explanatory. Notice that the equality relational operator is `==`, rather than `=`, which is the assignment operator. The meanings of the two operators really are different. The assignment operator causes the value of the variable on its left side to be the same as that of the quantity on the right; whereas the relational equality operator does not change the value of the quantity on either side. Since an assignment expression also has a value and could legally be used between the parentheses of an `if` statement, the C compiler has to know which operator you want. Using `=` for `==` is a common error of beginning C programmers, particularly those who have had some experience with other languages.

At this point, it might seem that the capability of the `if` statement is severely limited because it allows you to execute conditionally only a single statement. For example, suppose that, when calculating the absolute value, you wanted to keep track of the sign of `x` by setting an integer variable to either −1 or 1. The following code fragment looks like what you want, but contains a compile-time error.

```
/*
 * A flawed program fragment.
 */
    float x, y;
    int flag;

    if(x < 0.)
        y = -x;
        flag = -1;
    else
        y = x;
        flag = 1;
```

The compile-time error is that the `else` statement has to follow immediately the statement conditionally executed by the `if` statement. In the fragment, this statement is the `y = −x;` line. The compiler pays no attention to our indentation, and interprets the next line, `flag = −1;`, to be just the next statement in the program, which will be executed under all conditions. When the compiler discovers the `else` in the following line, it does not associate it with the preceding `if` statement and declares an error.

You could circumvent the compiler's concerns by rewriting the fragment as shown below.

```
/*
 * A flawed program fragment that runs but lies.
 */
    float x, y;
    int flag;

    flag = -1;
    if(x < 0.)
        y = -x;
    else
        y = x;
        flag = 1;
```

The idea here was that we would initially set `flag` to `-1`. If `x` is greater than or equal to zero, the `else` would be activated, and `flag` set to `1`. This program would compile and run, but it would not perform properly. The last statement, `flag = 1;` will be executed no matter the sign of `x` because only a single statement is allowed as the conditional block of the `else` statement.

Back to the original flawed fragment, to get the program to behave as desired, we have to make the compiler treat the two statements after the `if` as if they were one statement. (We also have to do the same for the two statements after the `else`.) We can do that by making use of the fact the the C compiler treats anything between curly brackets as a single statement. Thus, the following fragment would work as desired.

```
/*
 * Program fragment to set y = |x|,
 * and flag to 1 if x >= 0 or -1 if x < 0.
 * This one works!
 */
    float x, y;
    int flag;

    if(x < 0.) {
        y = -x;
        flag = -1;
    }
    else {
        y = x;
        flag = 1;
    }
```

In C a set of statements contained between curly brackets is called a *compound* statement, and it is common to form a single compound statement out of several single statements.

17.3.2 Program Flow and Loops

C has the analog of the `JMP` instruction of the PFW007. You can label a statement with any name you like, and the use the `goto` statement to jump there. As an example, the program in Fig. 17−4 is a C language version of the program in Exercise 15−1 which sums the integers from 1 to 6.

```
/*
 * Program to sum the integers between 1 and 6.
 */

main()
{
        int i, n, sum;

        i = 1;
        n = 6;            /* Upper limit of the sum */
        sum = 0;
loop1:
        if(i > n)         /* If i > n, we're finished */
            goto done;
        sum = sum + i;
        i = i + 1;         /* Increment i and */
        goto loop1;        /* go through the loop again */

                           /* Print out the result and go home */
done:
        printf("Sum from i=1 to 6 is %d\n", sum);
        return 0;
}
```

Figure 17−4. Simple program analogous to the PFW007 program to be written for Exercise 15−1 which sums the integers between 1 and 6.

To label a statement, simply assign it a name of your choosing and put the name, followed by a colon, before the statement. Often, such statement labels are placed on the line just preceding the labeled statement. (It's all the same to the C compiler because the compiler does not recognize the end of a line as terminating a statement.)

This program works, but is rather awkward. The main problem is the `goto`'s. It is generally best to

minimize the use of `goto`'s. Usually, but in my opinion not always, the use of `goto`'s makes the program harder to read. In some circles the use of a `goto` is considered a serious breach of etiquette!

Loops are pretty common features of programs, and C has two commonly-used ways of making them without using `goto`'s. The first uses the `while` statement. The program in Fig. 17−5 does the same thing as that in Fig. 17−4, but it is implemented with a `while` loop.

```
/*
 * Program to sum the integers between 1 and 6.
 * The program serves the same purpose as that in
 * Fig. 17-4, but this one uses a while statement
 * to implement the loop
 */

main()
{
    int i, sum;

    i = 1;
    sum = 0;

    while(i <= 6) {        /* The head of the while loop */
        sum = sum + i;
        i = i + 1;         /* Increment i */
    }                      /* The tail of the while loop */

                           /* Print answer and quit */
    printf("Sum from i=1 to 6 is %d\n", sum);
    return 0;
}
```

Figure 17−5. Modification to the program in Fig. 17−4 which uses a while loop.

Notice the use of a compound statement to form the body of the loop. Similarly to `if` statements, the `while` statement controls execution only of the statement following it. If you want more than one statement included in the loop use curly brackets to group them all into a compound statement.

As an aside, the operation of adding one to a variable, as is accomplished in the line `i = i + 1` in the program, is a very common one and C has a special operator for doing so. The line could have just as well been written `i++` or `++i`. Usually the placement of the `++` operator makes a difference, but in this case it happens that it doesn't. If you place the `++` before the variable C increments the variable before using it; whereas if you put it after, C increments it afterwards. In this particular case, the value of `i` is not used in the line in which it is incremented, so the order makes no difference. C also provides an autodecrement operator, `--`. The same rules about placement are followed.

To provide an example, I'll write the body of the `while` loop in the program we have been considering a little differently. It could have been written

```
while(i <= 6) {
    sum = sum + (i++);
}
```

Now the body of the `while` loop contains only a single statement, so the curly brackets are not necessary. I included them just for emphasis. In the body of the loop, the current value of `i` is added to `sum`, *and then* the value of `i` is incremented. If I had written the line with the `++` on the other side of `i`, the value of `i` would have been incremented *before* it was added to `sum`. In that case a different result would have been produced. One other thing: the parentheses around the `i++` term are not required, but I included them to make the line a little clearer.

Another loop structure which is used probably more commonly than the `while` loop is the `for` loop. Our sample program could be written using this loop as in Fig. 17−6.

```
/*
 * Program to sum the integers between 1 and 6.
 * The program is identical to that in Fig. 17−5,
 * except that it uses a for loop rather than a while.
 */

main()
{
    int i, n, sum;

    sum = 0;
    n = 6;                          /* The upper limit of the sum */

    for(i=1; i <= n ; i = i+1)
        sum = sum + i;              /* The body of the loop */

                                    /* Print result and quit */
    printf("Sum from i=1 to 6 is %d\n", sum);
    return 0;
}
```

Figure 17−6. Modification to the program in Fig. 17−4 which uses a `for` loop.

A `for` statement consists of the keyword, `for`, followed by a set of three terms, separated by semicolons and enclosed between the parentheses as in the figure. The first term is a statement which is executed only once, before the loop is started. It sets an initial condition. In this case, I used the term to set the initial value of `i` to `1`. The second term is a conditional giving the requirements for continuing the loop. It is checked just before each iteration of the loop, and the loop is executed again only if the value of the expression is non-zero. If the value is zero, iteration of the loop terminates and the program continues with the next statement after the loop (the `printf` statement in this case). The third term is a statement which is executed at the end of each iteration of the loop. In this example, this term was used to increment `i`. We could have just as well left this term blank, and put the statement `i = i + 1` at the end of the loop (just after the `sum = sum + i;` statement). Alternately we could, and probably should, have made use of the `++` operator.

The body of the loop consists of the (possibly compound) statement following the `for`. If the body of the loop consists of more than a single statement use curly brackets to group these statements into a single compound statement, just as with `if` statements and `while` loops. In the example in Fig. 17−6, the body of the loop contains only a single statement, so the curly brackets were not required.

The `for` instruction must have three terms associated with it or the compiler will balk, but one or more of these terms can be blank. I don't recommend it, but the `for` loop in Fig. 17−6 could also be written

```
    i = 1;                          /* Set the initial condition */
    for( ; ; ) {
        if(i <= n)                  /* Do this iteration? */
            sum = sum + (i++);      /* Yes, update sum, increment i */
        else
            goto done;              /* No, jump out of the loop */
    }
done:
```

17.4 Two Examples

In this section, we'll write two programs. Both are similar to programs we wrote for the PFW007. The first is almost an exact translation of the program we wrote for the PFW007 to multiply two numbers. The multiplication is done using shift and add instructions, rather that using the multiplication operator of C. The second program manipulates character strings. It prints out a character string backwards. We did not write a PFW007 program to do this specific task, but we easily could have. The manipulation of data stored in a character string is very similar to what we did with the PFW007.

17.4.1 Multiplying Integers "by Hand"

In this subsection, we will write a program that multiplies two integers using the same algorithm we developed for the PFW007. This is definitely the hard way to do it because C and the 80386 microprocessor would do it all for us if we just used the multiplication operator, `*`, but the program illustrates several features of C, and it illustrates the power of C to program the CPU at a very basic level. The program is almost a direct translation to C of the assembly language program for the PFW007 shown in Fig. 15−11.

The algorithm is essentially the same one you learned in school to do multiplication of numbers in base 10, except that it is applied to do multiplication in base 2. There is one simplification, though. To use the algorithm in base 10, you have to memorize a multiplication table of the products of all the numbers between 0 and 9. Similarly, to use the algorithm in base 2 you need the multiplication table for all numbers between 0 and 1, but that's a lot easier! The product of zero and any number is zero, and the product of 1 and any number is that number. I discussed the algorithm in Section 15−4, so I won't go into it in any more detail again. I suggest you reread this section now to familiarize yourself with it. Fig. 17−7 shows the program.

```
/*
 * Program to multiply two unsigned integers "by hand", as
 * in the program for the PFW007.
 */

main()
{
        int mpee, mpor, sum, top, bot;

        mpee = 26;
        mpor = 13;
        sum = 0;
        top = mpee;
        bot = mpor;
        for(sum=0; bot; ) {
               if(bot & 0x01)
                     sum = sum + top;
               top = top << 1;
               bot = bot >> 1;
        }

        printf("%d times %d:  Calculated %d, should be %d\n",
               mpee, mpor, sum, mpee*mpor);
        return 0;
}
```

Figure 17−7. Program to multiply two integers using binary multiplication at the bit level. The program is similar to the program developed for the PFW007 shown in Fig. 15−11.

There are two new features of C in this program which we haven't encountered before, and I have to discuss them before we can consider the details of the program. The condition in the if statement, `if(bot & 0x01)`, contains the first new item I want to discuss. The new feature here is the use of the ampersand, `&`. In this context, it means to bit-wise AND the numbers on either side of it. The operator

operates very similarly to the AND instruction of the PFW007, and the purpose of this line is to check whether the rightmost bit of bot is 1 or 0. The & operator uses the same symbol as the "address of" operator we have already discussed. Surprisingly, this dual meaning does not seem to cause much confusion.

The second feature is related to the << operator seen two lines down from the if statement. This is the left shift operator, and it means to left shift the number on its left the number of times given by the number on its right. The instruction top = top << 1; causes the number in top to be shifted left one bit. The next line contains the right shift operator, >>. It works just like the << operator, except that it shifts right instead of left.

C has several bitwise operators, and Table 17−2 shows all of them. All operators except the exclusive OR operator have exact analogs in the instruction set of the PFW007.

BITWISE OPERATORS in C

		Example		
Operator	**Meaning**	a	b	**result**
a & b	AND	00001111	01010101	00000101
a | b	OR	00001111	01010101	01011111
a ^ b	Exclusive OR	00001111	01010101	01011010
~a	NOT	00001111		11110000
a << b	Left Shift	00001111	00000010	00111100
a >> b	Right Shift	00001111	00000010	00000011

TABLE 17−2. The bitwise operators available in C. For clarity, the values of the variables a and b and the result are shown in binary.

We are now ready to consider the multiplication program. Five int's are declared, mpee, mpor, sum, top, and bot. The first two hold the two numbers to be multiplied, and the third will contain the sum as it accumulates. When finished, it should contain the answer. According to the algorithm, we have to left shift the multipliee and right shift the multiplior as we go through the multiplication process. Doing that modifies these variables, so I decided to declare two additional variables, top and bot. The idea is to copy the values of mpee and mpor to these variables, and then to manipulate top and bot rather than mpee and mpor. That way, the original values remain undisturbed, and can be printed out at the end.

The real work is done in the for loop. The if statement checks to see if the rightmost bit is one or not, and adds the value of top to sum if the bit is one. The value in top is then left shifted one place, the value in bot right shifted, and the process is repeated until all the bits in bot have been shifted off the right end. Finally, the program prints out the values of the two original numbers, the result of the algorithm, and what the result should be.

The values of mpee and mpor are set inside the program by "hard wiring" them to 26 and 13 respectively. As long as these are the only two values we want to multiply, that's fine, but it's pretty inconvenient to try other values. The program would be much better if there were some way to have it ask what values to use, and then read them in from the keyboard. There is a function, scanf which is standardly supplied with the C compiler, which can be used for this purpose. The function is similar to the printf function, except that it reads data into the program rather than printing it out from the program. Unfortunately, I cannot discuss it here very conveniently because to explain how it works requires some features of C I have not yet discussed. I'll discuss the use of scanf in some detail in the next chapter. For now, if you would like to play with using different values in the multiply program you might try replacing the first two executable statements (which set the values of mpee and mpor) with

```
printf("Enter mpee and mpor:  ");
scanf("%d %d", &mpee, &mpor);
```

The first line of the replacement just tells the user what the program is expecting. The second line reads in the two values entered and places them in `mpee` and `mpor`. Notice the ampersand before the two variables. In this context it is interpreted as the "address of" operator.

17.4.2 Reversing a Character String

In this section we'll write a program which prints out a character string in reverse order. First, we need a plan of attack—an algorithm. For clarity, consider a specific character string, say "University of Nebraska-Lincoln". Here's the plan I came up with. First, I'll load the string into an array of `char`'s. Then I'll find the address of the last character in the string, and set a pointer to point there. Finally, in a for loop, I'll print out the character in the location the pointer points to, decrement the pointer to point the the location just to the left, print out the character found there, and so forth until all the characters in the string have been printed.

To carry out the plan, there's only one new C feature I need to tell you about. How do you tell `printf` to convert the value in a variable to the ASCII character it stands for and then print out that character? The answer is the format specifier `%c`. You will see how it is used in the program. Notice the difference between the `%s` specifier that we've encountered before, and the `%c` specifier. Fig. 17−8 shows the program I came up with.

```
/*
 * Program which prints out a character string in reverse order
 */

main()
{
        char *str, *p;

        str = "University of Nebraska-Lincoln";

/* First find end of the string */
        for(p=str; *p; p++)
             ;                              /* No instruction */

/* p now points at terminating 0 byte */

/* Print out the characters, one at a time */
     for(p--; p>=str; p--)
          printf("%c", *p);

/* Print a new line */
        printf("\n");
        return 0;
}
```

Figure 17−8. Program that prints out a character string backwards.

The program starts by declaring two pointers to a `char`, `str` and `p`. Then it sets `str` to the starting address of the character string. We know where the string starts, and now the problem is to find where it ends. This is accomplished in the first `for` loop. Notice that the statement is immediately followed by a colon, meaning that there are no instructions in the body of the loop. The loop starts by setting `p` equal to `str` so that `p` points to the start of the string, and then it checks to see if the byte stored at that location is zero. If it is, the loop terminates; if it isn't, `p` is incremented so that it points to the next character in the string. Again, this byte is checked to see if it's zero, and if it's not the process is repeated. In this manner,

`p` steps through the character string until it finds the end, as indicated by a zero.

When the loop terminates `p` points to the terminating zero of the string; the last character is one byte before. Now, all that's left is to step backwards through the string, printing out each character, until we reach the start of the string, marked by `str`. This is accomplished by the second `for` loop. First, `p` is decremented so that it points to the last character of the string, rather than the terminating zero. If `p` is larger than `str` then we must still be to the right of the start of the string, and the character found there is printed out in the `printf` statement. The value of `p` is decremented, and the process repeated until `p` is less than `str`, implying that it points to the space just before the start of the string. At that point we're done.

The program runs, and prints out the following line

```
nlocniL-aksarbeN fo ytisrevinU
```

The program could be jazzed up by having it print out the string before reversal, but I'll leave that as a homework problem. Another possible improvement would be to make lower case the capital letters at the ends of the words in the reversed string, and the make the first letters of these words upper case. In other words, the modified program should print out `Nlocnil-Aksarben fo Ytisrevinu`. That's not as easy, but you know enough C to be able to write the program.

Just as with the previous program, the data to be operated on (the character string in this case) must be "hard wired" into the program because we have not discussed how to use the `scanf` function. If you are interested, I'll just tell you how to modify the program to use the `scanf` function. We'll talk about what's going on in the next chapter. The declaration statement and the next statement that follows it must be changed to

```
char str[100], *p;

printf("Enter the string to be reversed\n");
scanf("%[^\n]", str);
```

The first line declares `str` to be a pointer to a `char`, allocates 100 bytes of memory for storing the string, and sets `str` to point to the start of this memory block. We don't know how long a character string the user will enter, so we guess that it will be no longer than 99 characters. If that guess turns out to be wrong, the program will probably mess up. In the original program, we didn't have to allocate memory for the string explicitly because the compiler knew what the string was to be and allocated the memory automatically. The second line reminds the user what the computer wants, and the third actually reads in the string and stores in the memory block pointed to by `str`. Note that in this case there is no ampersand before `str`. Don't even ask about the first argument here!

17.5 Rear Window III: A Program in the Raw

In this section, we'll write a program which prints out in hex, byte-by-byte, the contents of the part of memory containing the first part of the program itself. This will be the machine language program that the CPU sees. My program will make use of the fact that the name of a function when used by itself is a pointer to the start of the function. Thus, `main` is a constant set to point to the start of the main program. Our program will put this value into a pointer variable, and then enter a loop in which it prints out the contents of the memory byte pointed to and then increments the pointer by one byte. As with the programs in section 17−1.1, the compiler must be set to the Tiny model for this program to work properly.

The program is shown in Fig. 17−9. Notice the use of the nested loops. The inner loop prints out the individual bytes. The variable `j` used to count the bytes, and 16 bytes are printed on a line. The outer loop prints out entire lines, and `i` is used to count the lines printed. The `printf` call inside the nested loops has a new twist in the format string. The string is `"%2.2x "`, and the `2.2` appearing before the `x` is the

```
/*
 * Program which prints out in hex the first 320 bytes of the
 * machine language version of itself.
 */
main()
{
    unsigned char *p;
    int i, j;

    p = main;
    printf("main starts at %x\n", p);
    for(i=1; i<=20; i++) {
        for(j=1; j<=16; j++)            /* Print out a line */
            printf("%2.2x ", *p++);     /* Print out a byte */
        printf("\n");                  /* Skip to next line */
    }
    return 0;
}
```

Figure 17−9. Program which prints out the machine language version of itself.

new twist. It was added to improve the appearance of the print out. The `2` before the period says that the printed value should use at least two spaces, whether needed or not, and the two after the period says that there should be two digits printed out, even if doing so requires leading zeroes. Finally, notice also the use of the increment operator, `++`, in the last argument of the same `printf` statement. Make sure you understand why it's placed after the variable, rather than before.

Running the program results in the output shown in Fig. 17−10.

```
main starts at 311
55 8b ec 4c 4c 56 57 c7 46 fe 11 03 ff 76 fe b8
80 14 50 e8 47 02 59 59 bf 01 00 eb 28 be 01 00
eb 15 8b 5e fe ff 46 fe 8a 07 b4 00 50 b8 93 14
50 e8 29 02 59 59 46 83 fe 10 7e e6 b8 9a 14 50
e8 1a 02 59 47 83 ff 14 7e d3 33 c0 eb 00 5f 5e
8b e5 5d c3 55 8b ec 83 3e 9c 14 20 75 05 b8 01
00 eb 13 8b 1e 9c 14 d1 e3 8b 46 04 89 87 e0 16
ff 06 9c 14 33 c0 5d c3 c3 55 8b ec eb 0a 8b 1e
9c 14 d1 e3 ff 97 e0 16 a1 9c 14 ff 0e 9c 14 0b
c0 75 eb ff 76 04 e8 5f fe 59 5d c3 55 8b ec 56
57 8b 76 04 0b f6 75 05 e8 73 00 eb 6b 39 74 0e
74 05 b8 ff ff eb 63 83 3c 00 7c 29 f7 44 02 08
00 75 0a 8b c6 05 05 00 39 44 0a 75 16 c7 04 00
00 8b c6 05 05 00 39 44 0a 75 08 8b 44 08 89 44
0a eb 35 eb 33 8b 44 06 03 04 40 8b f8 8b 04 2b
c7 89 04 57 8b 44 08 89 44 0a 50 8a 44 04 98 50
e8 1b 05 83 c4 06 3b c7 74 0e f7 44 02 00 02 75
07 81 4c 02 10 00 eb 9a 33 c0 5f 5e 5d c3 55 8b
ec 4c 4c 56 57 c7 46 fe 00 00 bf 14 00 be a4 14
eb 12 f7 44 02 03 00 74 08 56 e8 5f ff 59 ff 46
```

Figure 17−10. Output of program in Fig. 17−9.

This is the set of instructions in machine language which the CPU executes at the start of the program. Perhaps this gives you a greater appreciation of just what a help assemblers and compilers are to the

programmer! The first seven bytes specify six instructions which are used to set up the processor for executing the program. The first executable C statement, `p = main;` generates a `MOV` instruction, which in machine language is `c746fe1103`. In case you are interested, the dis-assembled translation of the first 30 or so bytes of the program is in Fig. 17−11, along with an indication of which C language statement generated them.

```
55            PUSH      BP
8BEC          MOV       BP,SP
4C            DEC       SP
4C            DEC       SP
56            PUSH      SI
57            PUSH      DI
C746FE1103    MOV       WORD PTR [BP-02],0311  ;p = main
B81103        MOV       AX,0311                ;printf(...
50            PUSH      AX
B88014        MOV       AX,1480
50            PUSH      AX
E84702        CALL      056F
59            POP       CX
59            POP       CX
BF0100        MOV       DI,0001                ;for(i=1;...
EB28          JMP       0357
BE0100        MOV       SI,0001                ;for(j=1;...
```

Figure 17−11. The first few instructions of the machine language program in Fig. 17−9 translated into assembly language. The corresponding C language statements are indicated after the semicolons.

The instruction set of the 80386 is considerably more complex than that of the PFW007, so you probably won't understand what several of the instructions do, but you should be able to get the general idea.

If you are interested in what assembly language statements the C compiler generates for any C program, you can cause the compiler to generate a file containing the assembly language version of any C program. I don't think you can do that from within the IDE. Instead, you have to get back to DOS and run the Turbo C compiler explicitly, including `-S` as an argument. Specifically, if I had put the program in Fig. 17−9 in a file named `F17−9.c`, then to produce an assembly language version of the program I would enter at the DOS prompt

```
tcc -S -mt F17−9.c
```

The `-S` argument tells `tcc` to make an assembly language file, and the `-mt` argument tells it to use the Tiny model. The result would appear in a file named `F17−9.ASM`.

17.6 Appendix: The Joy of Segmentation

In this appendix I'll try to tell you what's going on with all the "Models" the Turbo C compiler can use to generate code. At this point it would probably be a good idea to review the appendix to Chapter 14, section 14−8. The problems all arise from the way the Intel microprocessors use segmentation registers. Briefly, the internal registers of the 8086 were all 16-bit, but the processor was given a 20-bit address bus. Generating 20-bit addresses using only 16-bit registers is a problem which Intel solved by using segmentation registers. A 20-bit address is formed by left-shifting by four bits the contents of a 16-bit segmentation register and adding that to another 16-bit number.

A "feature" of the architecture is that different segmentation registers are used to form the addresses of different types of information. Program instructions use one segmentation register, named CS, data use another, DS, and the stack uses another, SS. When compiled with either the Tiny or Small models, the

addresses used in the programs in Figs. 17−1, 17−2, and 17−9 are all the 16-bit offset from the appropriate segmentation register. In the program of Fig. 17−1, the addresses were the offset from the stack register, SS, because the compiler put `a`, `b`, and `c` on the stack. In the program in Fig. 17−2 the addresses were offsets from the data register, DS, and in the program of Fig. 17−9 the address of `main` is the offset from the code register, CS. If you specify the Tiny model, all segmentation registers are set to the same value, so the offsets of all types of data all refer to the same origin. With the Small model, the segmentation registers are all loaded with different numbers, so that each of the three types of data has its own, non-overlapping, 64K bytes of memory.

With the other models, addresses of and pointers to one or more types of data are 32-bit numbers. The most significant two bytes contain the segmentation register value, and the least significant two contain the offset. The whole thing is a real mess! Fortunately, the 80386 and later members of the Intel family have 32-bit registers, and all this segmentation stuff is not necessary. Unfortunately, because of the sordid family history the mess will live on for quite some time.

17.7 Exercises

1. The goal of this exercise is to investigate the way floating point numbers are stored in computers using the 80386 microprocessor.

    a. Find out how many bytes are used to store a `float` and a `double` by writing a program in which two variables, `x1` and `x2`, are declared to be `float`'s, and two more, `y1` and `y2`, are declared `double`'s. Investigate how the variables are stored by having the program print out the addresses of all the variables. I suggest you declare both `float` variables in the same statement, one right after the other, and ditto for the `double`'s. From your results, how many bytes does the compiler use to store `float`'s and `double`'s? Explain how you arrived at your conclusion.

    b. What is the single-precision, IEEE standard floating point representation for the number (in decimal) 12? What is the double-precision representation? Give your answer in hexadecimal. Modify the program written in the previous part so that it sets one single precision variable and one double precision variable equal to 12, and then prints out in hex, byte-by-byte what is stored in memory for each variable. Reconcile what your program prints out with your answer to the first part of this question.

2. Verify that the program fragments (the two which put the string "Hello" into the array name `greeting`) at the start of section 17−2 work by writing a program which includes both fragments, and prints out `greeting` using the `%s` format specifier just after each fragment.

3. Write a program which encodes a character string by printing in order, all the odd-numbered characters first, followed by the even-numbered characters. For example, the string `engineer` would be changed to `egnenier`. Your program should print out the result. Run your program using the character string "Go in fear of abstractions".

4. Write a program which converts all letters in an ASCII-coded character string to upper case. There is a function (actually it's a macro but I won't go into that here), called `toupper` that comes with the C compiler which would do that job, but you are not to use it in this problem.

    HINTS: (See Table 15−1.)

    > If a number is the ASCII code for a lower case letter, it must have magnitude between `0x61` and `0x7a`.
    >
    > A lower case character can be converted to upper case by subtracting `0x20`.

5. Modify the program in Fig. 17−8 so that it prints out something like "The original string is", followed by the string on the next line, followed by a blank line, followed by "The reversed string is", followed by the reversed string on the next line.

6. In Exercise 15−7 you were to write a program for the PFW007 that encoded a character string by adding a fixed constant to the ASCII code for each character. That scheme had the problem that the output could have some strange control characters and some characters that are not legal ASCII codes (values larger than 127) in it. In this exercise you are to write a C program which is more sophisticated. Here's the scheme. Imagine writing all the upper case letters written in a circle in alphabetical order, going clockwise. Then, starting with A, the sequence would be A, B, C, ⋯, Y, Z, A, B, ⋯. To encode a given upper case letter, replace it with the letter *n* places further around, where *n* is a number that you choose. Lower case letters are to be handled similarly. The encoding scheme should change only letters—punctuation, numerals, and control codes should be left unchanged. For example, if *n* is 4, then the string "February" would be encoded to "Jifvyevc". Write a program which encodes character strings using this technique, and prints out the result.

Experiment with encoding whatever character strings you like, using whatever values of n you like, but turn in a program which encodes the character string "The value of R1 is 245 Ohms." using $n = 13$.

7. There are five students in a class, and they got the following scores on a test.

   83, 68, 73, 88, 96

   Write a program which stores these values in an array of `int`'s, and then calculates the average and the mean squared error of these scores. We discussed what the mean squared error in section 6.6, but in case you don't remember, the mean squared error, MSE, is given by

   $$MSE = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2$$

   where $\bar{x}$ is the average value of x, and N is the number of x's to be averaged. The program should print out the results for both the average and the mean squared error. Ideally, you would write the program so that the scores to be averaged would be read in when the program was run, but we haven't discussed how to do that yet so instead you will have to "hard-wire" the scores into the C program.

8. Write a program which calculates the factorial of a number. The program should set the value of a variable called `n` to this number, and you should be able to calculate the factorial of any number by simply setting the value of this variable to the desired number. Ideally, you would write the program so that the number would be read in when the program was run, but we haven't discussed yet how to do that so instead you will have to "hard-wire" the number into the C program. In section 15−9 we wrote a program for the PFW007 that calculated factorial using a recursive subroutine. C allows recursive subroutines, but you are not supposed to know how to do that yet, so write a program that works without using recursion. Test your program works by using it to calculate the factorial of 3 and 10, and comparing the output to the correct result.

9. Write a program that performs integer division at the binary level, similar to the multiplication program in section 17−4.1. The program should print out two integers, the quotient and the remainder. For example, dividing 52 by 11 should yield 4, remainder 8. Your program can use left and right shift operators and addition or subtraction, but it should not use the division operator. Test your program.

18. PROGRAMMING IN C TO SOLVE PROBLEMS

In this chapter we will concentrate on some useful features of C. We will not discuss all the features of C, but the subset we will discuss will be sufficient allow you to do just about anything that can be done with C, although maybe not in the most convenient way. If you are interested in learning more about the features not covered here, you should be able to do so pretty easily with standard books on the subject.

18.1 Functions

The official name for a subroutine in C is *function*, although most people seem to use the two terms interchangeably. Unless you specify otherwise, all functions return values. Such functions work similarly to many of the subroutines we wrote for the PFW007. When finished, they leave the result of their labors some place where the calling program can find it. For the PFW007 subroutines, we often left the result in register `R2`. It wasn't a problem with the PFW007 because it had only one data type (16-bit integer), but with C the return value of a function can be any of several possible data types (`short`, `int`, `long`, `float`, `double`, etc.) and the compiler must know what data type every function returns. If for any function you neglect to provide this information, the compiler assumes the function returns an `int`. I'll tell you how to specify a return value type in the next subsection.

The name of a function used by itself is a pointer to the start of the function in memory. We made use of this fact in the program in Fig. 17-9 to find the start of the main program. You call a function by using the name followed by the arguments enclosed in parentheses. As an example remember our use of the `printf` function in the previous chapter.

A function can have zero or more arguments. The values of these arguments are passed to the function by copying them onto the stack. Because of this way of handling arguments, a function is always dealing with a *copy* of its arguments, rather than with the actual arguments themselves. That means that a function cannot alter the value of any of its arguments. It can alter the value of the copy on the stack, but that's lost anyway when the function returns. In case you find that confusing, here's an analogous situation. Suppose you are writing an essay and you have some misspelled words in it. You write the words on a piece of scratch paper, go to the dictionary with the scratch paper, find the correct spelling, and correct the words on the scratch paper, but do not return to your essay to make the changes there. The originals in the essay remain unchanged, and you haven't done anything useful. The arguments to a function are analogous to the words on the scratch paper, they are copies of the originals. If you only need the information they contain, that's fine, but if you want to change the originals, the copies are of no use.

At first glance, this would appear to be a real limitation of the C language, since sometimes you do want a function to modify one or more of its arguments. The solution is to pass the function not the variable to be modified itself, but rather its address. The function then knows the address to be modified, and it can then write whatever it wants into the memory location given it. It does not matter that the address is a copy because it is not the address that is to be modified, but rather the *contents* of the address. All this is probably most easily explained with examples, and I discuss two here.

18.1.1 A Function which Returns a Value.

The first example program reads a character string from the keyboard, up to a newline character, stores the string in an array of `char`'s, and then prints out the string and the number of characters in the string. (The newline character is the character generated when you press **Enter** on the keyboard.) The program uses two functions which return useful values. One, `getchar()`, is supplied in the library that comes with all C compilers (similarly to the `printf` function). The other, `str_no()`, is a function I wrote.

The program is shown in Fig. 18−1. Space for storing the character string is allocated in the first declaration statement. This statement allocates a block of memory, and sets `line` to point to the start of the

```
/*
 *  Program which counts the number of characters in a string
 *  entered from the keyboard.
 */

main()
 {
        char line[100], *p;
        int c, n;
        int getchar(void), str_no(char *);

/*  Inform the user what's up and read the string */
        printf("Enter the character string.\n");
        p = line;
        c = 1;                  /* To start off while loop */
        while(c != '\n') {
                c = getchar();
                *p++ = c;
        }
                                /* Terminate line with a 0, */
        *--p = 0;               /* backspacing over the \n */

/*  Count the characters and report the count */
        n = str_no(line);
        printf("You entered:\n%s\n", line);
        printf("The line contained %d characters\n", n);
        return 1;
}

/*
 *  Function which counts the number of characters in a string.
 */

int str_no(char *s)

{
        int n;

        n = 0;
        while(*s++)
                n++;
        return n;
}
```

Figure 18−1. Program illustrating the use of functions. The function `str_no()` counts the number of characters in the character string pointed to by `s`, and returns the result.

block. It also declares `p` to be a pointer to a `char`. I will use this pointer to keep track of where in `line` the next character read in from the keyboard is to be placed. The next line declares `c` and `n` to be `int`'s. The following line tells the compiler about the two functions that will be called, `getchar` and `str_no`. Both functions return a value and we will be using these values, so we have to tell the computer what data type the functions return. This is the purpose of this line. It says that both functions return an `int`. It also says that `getchar` has no arguments (that's what the `void` means) and `str_no` has one, a pointer to a `char`. If you do not declare a function, you can still use it, but the compiler will assume that it returns an `int`, and will make some probably incorrect assumptions about the arguments. A mistake I frequently make is to use a function that returns a `double` (like `sin()`, or `log()`) in my program, and forget to declare it. The compiler assumes the function returns an `int`, and when the program runs, it completely misinterprets the value the function returns. I end up with a program that seems to run correctly, but produces strange results.

The program first prints out a request to the user to enter the string to be counted. It then sets `p` to point to the start of `line`. Ignore the next instruction, `c = 1;` for the moment. The character string is read in and stored in `line` in the body of the `while` loop. The characters are read in using the function `getchar()`. This is a standard function which is supplied with all C compilers. (That means I don't have to write it; someone else already has.) This function waits for the user to type a character at the keyboard, and then returns the ASCII code for it. If you type several characters, each call to `getchar()` gets just one, in the order typed.

For every iteration of the `while` loop, the compound statement enclosed in the curly brackets is executed. The first statement of this compound statement gets the next character from the keyboard, and stores it in `c`. Next, the character is copied into the array, `line`, with the instruction `*p++ = c;`. The variable `p` is a pointer to a `char`, which was set equal to the address of the start of the memory block assigned to the array named `line` before entering the loop. This statement copies the character stored in `c` to the location pointed to by `p`, and then increments `p` to point to the next location in `line`.

After finishing this instruction, the loop iterates again. First, it checks to determine if the last character read in was the newline character, `\n`. If it was not, the loop is executed again. If it was, the statement following the closing curly bracket of the loop, `*--p = 0;` is executed. The purpose of this statement is to terminate the character string in `line` with a 0 byte so that it conforms to the C convention for character strings. There's one little complication—the `while` loop copied all the characters entered, including the newline character resulting from pressing **Enter**. We probably don't want to include the newline in the count, so `p` is first decremented in order that the zero will overwrite the unwanted newline character.

The number of characters in `line` is counted in the last section of the program, and both the character string and the number of characters in it are printed out. The counting is done in the function I wrote and named `str_no`, which is defined at the bottom of the listing after the closing curly bracket for `main`. This function has one argument, and it returns the number of characters it counts. The argument is a pointer to the start of the string to be counted. I'll discuss how `str_no()` works shortly, but first I want to finish discussing the main program. The line `n = str_no(line);` performs two duties. First it calls my function, giving it the address of the character string to be counted, and second it puts the value my function returns into the variable `n`. Finally, the program prints out the character string it just read, followed by the number of characters in it.

Before continuing, I have to take care of a little old business. What is the purpose of the line `c = 1;` near the start of the program which I told you to ignore? The answer is it ensures that the following `while` loop gets started off correctly. The first thing the `while` does is check its conditional. Without the statement in question, the value of `c` is not set when the loop is entered, and the value of the conditional is indeterminate. I want the body of the `while` to be executed at least once, so I initialize `c` to a value that ensures the value of the conditional will be 1.

With that out of the way, I'd like to discuss my home-brew function `str_no()`. How does it work, and how do I tell the C compiler about it? To answer the second question first, the definition of `str_no()` starts with the line `int str_no(char *s)` This line tells the compiler that the code between the following matched curly brackets is a function to be known as `str_no` which returns an `int`, and that the function has one argument, a pointer to a `char`, which is known inside the function as `s`.

The variable `n` is declared to be an `int` in the first line inside the matched brackets. This variable is to be used to count the characters in the string, so it need only be known inside of the function. Indeed, it is best that `n` be isolated from the outside. It is quite possible that a programmer could inadvertently use the same name for a different variable in the main program, or maybe in another function. If the name of `n` were recognized everywhere, these variables would share the same memory space, and modifications to the `n` inside `str_no()` would also change the value of the other variables with the same name. That is an undesirable behavior, particularly for general purpose functions like `getchar()` which are written by one person and used, without source code, by others. Suppose, for example, that the writer of `getchar()` had

used the variable name `p`. If that variable were known outside of `getchar()`, then the part of our main program containing the body of the `while` loop would almost certainly misbehave because `getchar()` would modify `p` without our knowledge.

For this reason, variables declared inside a function are considered local to that function. If we want a variable to be known to two or more functions, we should place the declaration for the variable outside any function, and before the first in which it is to be recognized. Such a declaration is said to be *external*, and if it is placed before all function definitions, including `main()`, it is also *global*. Variables declared inside a function are given automatic storage. That means that the memory space for them comes from the stack, and that the variable effectively disappears when the function exits. If the function is called a second time, it is unlikely that the variable will have the same value on entry as it did when the function exited the previous time. There are two ways to force a variable to retain its value after a function exits. One way is to declare it outside any function, so that it is an external variable. The other way is to use the `static` keyword with the declaration inside the function. For example, the declaration for `n` in `str_no` would be `static int n;`. In this case, `n` would be known only inside `str_no`, but its value would remain intact after `str_no` exits, and we could access it again in another call to `str_no`.

Back to how `str_no()` works, `n` is to be used to count the characters, and is first initialized to `0`. Then, within the `while` loop `n` is incremented once for each character in the character string. Notice that the `while` detects the end of the string by detecting the terminating zero byte. Notice also, that the conditional for the `while` "kills two birds with one stone" in that it both checks for the end of the string, and it increments `s` to point to the next character in the string. When the loop terminates, the line `return n` causes the function to return the value of `n` (which by that time should be the number of characters in the string). Recall we encountered the `return` statement previously as a magical way to keep the compiler from griping at us. Maybe now the usage will make a little more sense.

I have several comments about this program. First notice that the value of the argument `s` is changed within the `str_no()` function, and that `str_no()` is called with `line` as an argument. Changing `line` is a high no-no because it points to the start of the memory block allocated in the declaration statement. If you change that, you will have lost the address of this block with no way to retrieve it. Everything is OK, though, because `s` is a copy of `line`, rather than `line` itself. When the function returns, the value of `line` will remain unchanged—only the copy on the stack is different, and that's lost anyway when the subroutine returns. (Remember that both `s` and `line` contain the address of the string, rather than the string itself.)

The `while` loop in the main program in which characters are read could be rewritten as follows

```
while((c=getchar()) != '\n')
        *p++ = c;
```

In this case, the conditional for the `while` does double duty. First it gets the next character and stores it in `c`, and second it checks to see if the character was a newline. This funny business works because an assignment statement is itself an expression with a value. That value is just the value of whatever was assigned. In this case, the value of the expression `(c=getchar())` is whatever character `getchar()` returned. The parentheses enclosing the statement in the conditional are required. Without them, the compiler would interpret the expression as `c = (getchar() != '\n')`, which assigns to `c` the result of comparing the return value of `getchar` with `\n`. In that case, `c` would be either `0` or `1`, depending on whether the return value was `'\n'` or not.

With this change, the statement `c = 1;` just preceding the `while` loop is not needed because the character is read before comparing `c` to `'\n'`. Also, the statement just after the loop must be changed to `*p = 0;` because `p` is not incremented when `getchar()` returns a newline.

This minimalist trend could be continued. The `while` loop could be reduced to just one statement, and the variable `c` eliminated. The loop in `main` could be reduced to

```
while((*p++ = getchar()) != '\n') ;
```

In this case, the conditional for the loop stores the return from `getchar()` in `*p` directly, and the increments `p`. Additionally, the variable `n` could be eliminated. In this case, the second `printf` statement would be changed to

```
printf("The line contained %d characters\n", str_no(line));
```

and the line `n = str_no(line);` would be deleted. When the computer sets up to call `printf`, it needs to put the value of the second argument on the stack. To do that, it calls `str_no()` and pushes the value it returns onto the stack in the place where the second argument should be. `printf()` doesn't know the difference, and everything works the same.

18.1.2 ANSI vs K&R C

Before continuing, I need to take a little detour. There are two versions of C in common usage: K&R C (named after the guys who invented the language, Brian Kernighan and Dennis Ritche) and ANSI C (named after the American National Standards Institute). K&R is the older version, and for a time it was the standard. It achieved this status by default rather than by formal approval. More recently, a national committee has agreed on a standard C which is very similar to K&R C, but with several extensions. This is ANSI C. ANSI C differs from K&R C mostly in minor details, but there is one significant difference I want to discuss here. In K&R C, the declaration for the two functions doesn't tell the compiler what kind of arguments to expect. For example, in Fig. 18−1, one would have written instead for the line declaring `getchar` and `str_no` near the start of the program

```
int getchar(), str_no();
```

This older syntax is a little easier to write, but it can lead to some programming errors that are difficult to find. The compiler has no idea how many arguments each function expects, or what the expected type of the arguments are. Consequently, if you later call a function with the wrong number or type of arguments, the compiler will let you do it and make no complaint. The program may or may not run to completion, but the results will almost certainly be wrong. The floating point math functions like `sqrt()` provide a good example of the kind of trouble one can get into. The `sqrt()` function expects a `double` argument, and it returns a `double`. If it were declared with the K&R syntax as

```
double sqrt();
```

and you were to use it to calculate the $\sqrt{2}$, you might write

```
double answer;

answer = sqrt(2);
```

This would give a very strange result for `answer` because the compiler interprets `2` to be an `int`, and pushes the integer representation of the number 2 onto the stack as an argument for `sqrt`. When the `sqrt` function executes, it looks for its argument on the stack, finds the `int` representation of 2, and interprets this as the `double` representation of the number to be "square rooted." The representations of the two data types are quite different, so `sqrt` will translate the bits it sees into a very different number from 2, and give a nonsense result. If the `sqrt` call were imbedded inside a large program, it could be rather difficult to find the problem.

In order to avoid such problems, the ANSI committee added *function prototypes* to C. When a function is declared, you can tell the compiler how many, and what type of arguments it expects. With this information, it can convert arguments to the expected type, or issue a warning if it is not possible to do so. Back to our $\sqrt{2}$ example, if the `sqrt` function were declared

```
double sqrt(double);
```

the program would work correctly under ANSI C. Getting the function prototypes right is sometimes a nuisance, but I recommend using them. They can catch some pretty subtle bugs.

There is also a syntax difference when the function is actually defined, for example as near the bottom of Fig. 18−1. In K&R C, the ANSI C line

```
int str_no(char *s)
```

would be written as two lines

```
int str_no(s)
char *s;
```

When you call a function with arguments, it is important that each argument you supply is pushed onto the stack in the proper representation. If you use function prototypes in ANSI C, the compiler will pretty much take care of this for you (or at least issue a warning), but if you don't, you have to take care of it yourself. Similarly, you have to be careful to declare the correct type of return value for the function. If you get the wrong data type, the return value of the function will be misinterpreted.

The situation in which I make this mistake most often is when I forget to declare a function before using it. If you do not declare a function called in a program, you can still use it, but the compiler will automatically assume it returns an `int`. For the standard math routines like `sqrt()`, `exp()`, etc this is an easy mistake to make, and it always leads to difficulty. These functions all return `double`'s. If they are not explicitly declared before use, the compiler incorrectly assumes that they return `int`'s. The representation of a given number in the two formats is quite differents, so if I make this mistake, the function will return the correct value, but in a different "language" than the compiler expects. The program will misinterpret this result and obtain a very wrong answer.

This is a common problem, and the compiler comes with a file containing declarations for all the math functions. For the Turbo C compiler, the file is named `\TC\INCLUDE\MATH.H` and every time you use functions from the standard math library, you could copy this file into your program somewhere near the start. C has a feature which we'll discuss later in this chapter which instructs the compiler program to do that for you before it starts to compile the program. To use it you would place the following line near the start of your program.

```
#include <math.h>
```

We'll discuss the `#include` directive in more depth is Section 18−4.

18.1.3 A Function which Returns Information Via an Argument

Here's another example, a function, `avg()`, which averages the values in an array of `float`'s, and returns the result in one of its arguments rather than as a return value. The function and a main program to test it are shown in Fig. 18−2. A pointer to the array, the number of elements in the array, and the address of the memory location where the result is to be placed are passed to the function as arguments. The main program tests the function by making an array containing the first 100 integers, stored as `float`'s, and then calling `avg()` to calculate the average.

There are several points I want to discuss here. The main point is that the home-brew function `avg()` returns a result through one of its arguments. Recall that doing that is not straight-forward because the function only sees copies of the variables it's called with, so that if it modifies an argument, it only modifies the copy, not the variable itself. From the `main` program, we want the function to place the answer in the variable `z`. To do that, we pass the function not the value of `z`, but rather the address of `z`. Knowing the address of `z`, the function can write its return value to that address, modifying the value of the variable. It does that in the last statement of the function, `*ans_ptr = sum/n`. This is a pretty common procedure with C, and it's important that you understand what's going on.

Let's look at the program in detail. The `avg()` function returns all the useful information it has via an argument, so there should be no return value. The line declaring `avg()` near the start of the program tells this to the compiler. That's the meaning of `void`.

The declaration line in `main` allocating storage space for `x` allocates space for 100 `float`'s. These

```
/*
 *  Program to test a function which averages a set of
 *  floating point numbers contained in an array.  The
 *  program uses the function to average the first 100
 *  integers.
 */

main()
{
        int i, npts;
        float x[100], z;
        void avg(float *, int, float *);

/*  Put the integers 1 thru 100 in the array */
        npts = 100;
        for(i=1; i<=npts; i++)
                x[i-1] = i;

/*  Calculate the average, putting it in z */
        avg(x, npts, &z);

/* Report the answer and go home */
        printf("The average of the first %d integers is %g\n", npts, z);
        return 0;
}

/*
 *  Subroutine to average the n numbers in array and return the result
 *  via the argument ans_ptr
 */

void avg(float *array, int n, float *ans_ptr)

{
        int i;
        float sum;

        sum = 0.;
        for(i=1; i<=n; i++)
                sum = sum + array[i-1];
        *ans_ptr = sum/n;
}
```

Figure 18−2. Program illustrating a the use of function which returns a result via one of its arguments.

individual `float`'s are accessed by `x[0]` through `x[99]`, not `x[1]` through `x[100]` because the array index starts at 0, not 1. In both `for` loops the counter variable `i` goes from 1 to 100, so the index to the array is `i-1`, rather than `i`. Getting the range of array indices right is a detail I mess up frequently.

Finally, I have two comments about the code inside the `avg` function. First, consider the statement inside the `for` loop, `sum = sum + array[i-1];`. In this statement, the value of `sum` is to be modified by adding the value of `array[i-1]` to it. These kinds of operation are pretty common, and C has a family of operators for them. In this case, we could have written `sum += array[i-1]`. The `+=` operator tells the compiler to add the value of whatever is on the right to the variable on the left and store the result in the variable on the left. It's a kind of shorthand. You can use the same shorthand with any operator in C for which it makes sense: `-=`, `*=`, `/=`, `&=`, `<<=`, for example.

Second, notice that in the last statement `sum`, a `float`, is divided by `n`, an `int`. To perform that

operation, both operands must be of the same type. When such an operator connects variables of different types, the C compiler automatically does a conversion so that both have the same type, and then performs the operation. The compiler is pretty smart about deciding what to convert to what, so that almost always the result is what you had intended, but occasionally the compiler makes the wrong guess. For that reason, you need to be aware that the compiler does such conversions.

So what exactly does the compiler do with the line `*ans_ptr = sum/n;`? To do the division, does it convert the floating point variable to an integer, or the integer variable to floating point? Generally, the compiler chooses the option which loses the least information. In this case, it would convert the integer to a floating point value before doing the division.

18.2 Printf, Scanf, and Format Specifiers

18.2.1 Printf and Format Specifiers

We have already discussed the use of the standard `printf()` function to print stuff on the screen. `printf()` is simply a subroutine that someone has already written and compiled and made available with the C compiler. It's cleverly written so that it can accept an arbitrary number of arguments of arbitrary type. The first argument is required and must be a `char *`, pointing to a character string. It is this string that tells `printf()` how many additional arguments it has, and what the types of the arguments are. When called, `printf()` prints out this string almost verbatim, substituting the values of the arguments for the format specifiers. The first format specifier is replaced with the value of the first argument after the format string, the second with the second, and so forth.

A format specifier consists of the percent sign character, `%`, followed immediately by a few more characters which tell `printf` what type the corresponding argument is, and how to print it out. The most general form of a format specifier is a `%` sign, followed by a number, followed by a decimal point, followed by another number, followed by one or occasionally two letters. Specifically, a format specifier has the form `%w.pt` when `w` and `p` are integers, and `t` is one or occasionally two letters. The first number, `w` tells `printf` how much space to use to print the item out and the second, `p`, what precision or how many digits of accuracy to use. The letter(s), `t` tells `printf` what the type of the corresponding variable is, and how to print it out. Most of the possible letters are shown in Table 18−1.

CONVERSION TYPE SPECIFIERS

Format Specifier	Argument Type	Output
`c`	`int, short, char`	single character
`d`	`int, short, char`	signed integer, printed in base 10
`e`	`float, double`	floating point with exponent
`f`	`float, double`	floating point without exponent
`g`	`float, double`	floating point either with or without exponent, depending on size of number
`o`	`int, short, char`	unsigned integer, printed in base 8
`s`	`char *`	character string printed out
`u`	`int, short, char`	unsigned integer, printed in base 10
`x`	`int, short, char`	unsigned integer, printed in base 16

TABLE 18−1. Table showing the meaning of several common conversion type specifiers in the format string for `printf`.

You can precede the `d`, `o`, `u`, and `x` specifiers with a lower case `l` (that's the letter "*el*") to specify that the argument is a `long`. We have already encountered most of these specifiers. The difference between the

specifiers `e`, `f`, and `g` is not very clear from the table so a few more words of explanation are in order. Consider, for example, the number -1368.329. Using the `f` format, this would be printed `-1368.329`, whereas using the `e` format it would be `-1.368329e+03`. The `g` format may print the number in either `e` or `f` format. It chooses on the basis of how large the number is. If the number is very large or very small it would require many trailing or leading 0's to print in the `f` format, so `g` opts for the `e` format. For more moderate sized numbers the `e` format is the more clumsy and `g` opts for `f`.

The width and precision parts of a specifier, *w* and *p*, require more discussion. The width gives the minimum number of spaces to use in printing the item. If the item requires more spaces, more spaces will be used, but if it requires fewer spaces than specified, the printed item will be padded with blanks to make up the deficit. The precision part has a different effect on different types of items. For `d`, `o`, `u`, and `x` types the precision specifies the minimum number of digits to appear. If fewer digits are required than this minimum, the number is padded with leading zeroes. For `f` and `e` items, the precision specifies the number of digits to appear after the decimal point. If no precision is specified, it defaults to 6. For `g` items the precision specifies the total number of digits (excluding those in the exponent if present) to be printed. With the `s` specifier, the precision specifies the maximum number of characters to be printed. If the precision is missing in this case it's taken to be infinite. Fig. 18−3 shows the result of printing some numbers with several format specifiers.

```
Formats:
%d:1234           %u:1234           %o:2322           %x:4d2
%d:-1234          %u:64302          %o:175456         %x:fb2e
%f:0.000001       %f:1.234000       %f:1234000.000000
%e:1.234000e-06   %e:1.234000e+00   %e:1.234000e+06
%g:1.234e-06      %g:1.234          %g:1.234e+06


Field widths:
 %6d:  1234          %3d:1234
 %6d: -1234          %3d:-1234
%12f:    0.000001   %12f:    1.234000   %12f:1234000.000000
%12e:1.234000e-06   %12e:1.234000e+00   %12e:1.234000e+06
%12g:   1.234e-06   %12g:       1.234   %12g:   1.234e+06


Precisions:
 %6.6d:001234          %6.3d:  1234
%12.2f:        0.00   %12.2f:        1.23   %12.2f:  1234000.00
%12.2e:    1.23e-06   %12.2e:    1.23e+00   %12.2e:    1.23e+06
%12.2g:     1.2e-06   %12.2g:         1.2   %12.2g:     1.2e+06
%12.6f:    0.000001   %12.6f:    1.234000   %12.6f:1234000.000000
%12.6e:1.234000e-06   %12.6e:1.234000e+00   %12.6e:1.234000e+06
%12.6g:   1.234e-06   %12.6g:       1.234   %12.6g:   1.234e+06
```

Figure 18−3. Illustration of the behavior of `printf()` for several format specifiers. For the float specifiers three numbers are printed out on each line: 1.234×10^{-6}, 1.234, and $1.234\times10^{+6}$. For the integer specifiers, on the first line of each section 1234 is printed in several formats, and in the second line of the first two sections −1234 is printed.

18.2.2 Scanf

The standard function `printf()` is used to print out values of variables known inside a program. The standard C library contains an analogous function used to read values of variables into a program. The function is called `scanf`, and it has arguments which are similar to those used by `printf`. I used the `scanf()` function in Chapter 17 but was unable to explain very much to you about how it works. I now

can do better, and that is the purpose of this subsection. The function provides a very powerful data input capability, but it can be quite difficult to use some of the more esoteric features. I won't tell you everything about `scanf()` here, but there should be enough information to use it for most data input purposes.

The first argument to `scanf` must be present, and must be a character string. The string is a format string, and it follows conventions very similar to those used with `printf` formats. The format string tells the function how many and what types of variables are to be read in. The remaining arguments tell `scanf` where to place the values read in.

Because of the fact that C functions cannot modify the values of their arguments, you must supply `scanf` the address of the variable, rather than the variable itself. Thus, to read in the value of a variable, say `n`, declared to be an `int` you could use the line `scanf("%d", &n);`. When the program encounters this line, it will stop and wait for you to type something in from the keyboard. As you type characters in, they are stored and are not passed to your program until you press **Enter**. Doing that puts a newline character on the end, and signals your program that there is some input waiting for it. At that point, `scanf` will collect characters until it encounters something that could not be part of a signed, base 10 integer. It then stops collecting, puts the impostor character back on the input queue, converts the characters it has into an integer, and places the result in the location pointed to by `&n`.

`scanf()` is of two minds about whether or not a blank or other white space character can be part of a number. When it starts trying to get a number, `scanf()` considers blanks to be leading white space, ignores them, and keeps on collecting. Once it has seen a non-white space character, however, it considers a white space character to be illegal for a number, and assumes that the white space marks the end of the number. This behavior usually produces the result you would want because if you haven't started typing in the number you would probably want any white space characters to be ignored (they might be left over from the end of the last number you typed in). Once you have started on the number, the appearance of a white space character probably means that's the end of the number. Then you would want `scanf` to stop reading characters for the number.

If you want to read in the values of two variables, one an `int` named `iv` and the other a `float` named `fv`, you could use the line `scanf("%d %f", &iv, &fv);`. For the `float` variable, you could use either `f`, `e`, or `g` in the format string. They all produce the same result. When executed, the line would cause the computer to stop and wait for you to type in something. Suppose you typed `126 1.358` followed by pressing **Enter** (with two blanks between the two numbers). Then `scanf` would collect the first three characters, `126`, stopping collection when it encountered the first blank separating the two numbers, convert the character string `"126"` to the number 126, and store the number at the location provided by `&iv`. The format string tells `scanf` to get a second number, so it would start collecting characters again, this time for a `float`. The blank it encountered earlier which terminated the first number was put back on the input queue, so it would first read this blank, followed by another blank. These are leading white space, so they would be ignored, and character collection would continue. The characters `1`, `.`, `3`, `5`, and `8` are all legal parts of a `float`, so `scanf()` would collect all of them, stopping only when it encountered the new line character resulting from pressing `Enter`. It would consider that to be trailing white space, put it back on the queue, convert the character string it had gathered to a `float`, and put the result in the location pointed to by `&fv`.

One reason that `scanf()` is more difficult to use than `printf` is that it is very particular about its arguments. The types of all the arguments you supply must match with the types the format string tells `scanf` to expect because `scanf` sees the address of the variable rather than the variable itself. ANSI function prototypes don't help here. If you want to read a number into a variable declared to be a `double`, for instance, you have to tell `scanf` that the argument points to a `double`, rather than a `float`. Thus if `dvar` is declared to be a `double` you would use the line `scanf("%lf", &dvar);`. Notice the presence of the `l` (that's a lower case "*el*") before the `f` specifier to tell `scanf` that a `double` is expected. If you want to read a number into a `short`, and `short`'s and `int`'s are different on your machine, use `h` before the conversion type specifier; `%4.2hd` for example.

The `scanf` function tries to match characters in the format string other than reasonable specifiers and blanks exactly in the input. This capability is pretty powerful, but very tricky to use. I suggest you stick to separating format specifiers with blanks, and separating numbers you type in by blanks or new lines. I've spent many frustrating hours trying to get `scanf` to do what I wanted when I had more adventuresome goals for it. From this experience, I can offer some useful advice. If your program is misbehaving for no apparent reason, I suggest that you first look at any `scanf`'s that are in it.

18.3 Multi-Dimensional Arrays

In C you can use arrays of more than one dimension. You can declare multi-dimensional arrays in two ways, which I will cleverly call the first scheme and the second scheme. Referencing the arrays is the same for both, but the way the data is stored in memory can be different. As an example of the first scheme, consider the following program fragment.

```
char x[4][3], c;

c = x[2][1];
```

The declaratory statement tells the compiler that `x` refers to a two-dimensional array with three columns and four rows. It also allocates memory space for $4 \times 3 = 12$ `char`'s, and sets `x` equal to the starting address of this memory block. The next statement sets `c` equal to the value stored in the second column, third row (remember indices start at 0, not 1). The elements of `x` are stored sequentially in memory, with `x[i][j+1]` just after `x[i][j]`, and `x[i+1][0]` just after `x[i][2]`.

The second scheme to allocate multi-dimensional arrays is similar to the first, but it is more general. Continuing with our 4×3 example, but choosing instead the second scheme, we would allocate space for four one-dimensional arrays, each holding three values—one 1-D array for each row of the 2-D array. We would also allocate a 1-D array of four pointers to `char`'s to mark the start of each of the 1-D arrays corresponding to the rows. With this method, we must explicitly put the proper addresses in the pointer array. A clumsy way to implement this method is

```
char row1[3], row2[3], row3[3], row4[3];    /* The actual rows */
char *x[4];                                 /* Pointers to rows */
char c;
int i;

/* This sets up the array of pointers */
    x[0] = &row1[0];
    x[1] = &row2[0];
    x[2] = &row3[0];
    x[3] = &row4[0];

/* Just as before, we can access the 2,1 element by */
    c = (x[2]) [1];    /* c = x[2][1] works identically */
```

It is easier to allocate the memory for storage as one 12-byte block rather than as four 3-byte blocks as above. In this case the fragment is

```
char block[12];        /* This is the storage space */
char *x[4];            /* These elements point to the rows */
char c;
int i;

/*  This sets up the array of pointers */
    for(i=0; i<4; i++)
        x[i] = &block[3*i];
```

Fig. 18−4 shows how memory is allocated for both schemes (assuming that for the second scheme memory was allocated as a single block, as just above).

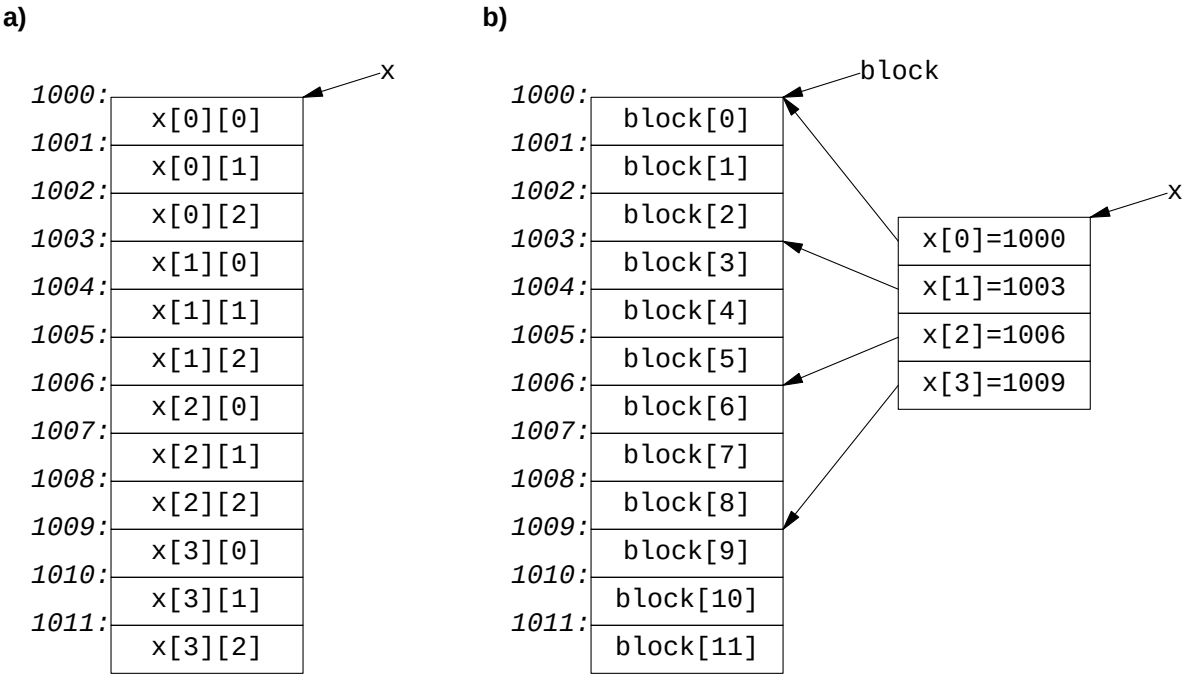

Figure 18−4. Schematic drawing illustrating the two schemes for allocating a two-dimensional array. Each rectangle corresponds to one byte of memory, and a possible set of addresses for each is shown (in decimal) just to the right of the top of each rectangle.

To help in the discussion, I've assumed that in both cases the block of memory holding the data is located in memory address 1000 through 1011, expressed in decimal. With the first scheme, the block is set aside and `x` is given the value 1000, the starting address of the block. To access `x[2][1]` the compiler must find the start of the third row by skipping over the space required for the two rows above it. Since there are 3 columns in the array, there are 3 elements in each row, and a row requires 3 bytes. (If `x` had referred to an array of `float`'s instead of `char`'s, each row would require 12 bytes because each element would require 4 bytes instead of just 1.) Thus the address of the start of the desired row is `x` + 2×3, or 1006. The desired element is the second in the row, so to find it the compiler must skip an additional space, making the address `x` + 2×3 + 1, or 1007. To carry out the statement `c = x[2][1];` the compiler first finds the address of the element on the right hand side using the procedure above and then it copies what it finds there into the memory space allocated to `c`. In general for an array with the declaration `char z[`*nrows*`][`*ncols*`]`, to access the *i*, *j*th element, `z[`*i*`][`*j*`]`, the compiler would look in the address given by `z` + *i*×*ncols* + *j*.

With the second method, two arrays are involved. The array labeled `x` is an array of four pointers to locations in the actual data storage area, labeled by `block`. The compiler automatically sets `block` to point to the start of the memory associated with `block`, 1000 in this case. It also sets `x` to point to the start of the array of `char *`'s associated with `x`, but it does not fill in any values in the `x` array. The programmer has to do this explicitly. We want the first element of `x` to point to the start of the memory holding the first row of the 2-D array, the second element to the block holding the second row, and so forth. Setting the first one is easy; we choose to start the first row at the start of the memory block. Thus in the hypothetical case in Fig. 18−4b, we would have to include explicitly the instruction `x[0] = 1000`. I chose to start the block holding the second row just after the end of the first, so I would have to include the instruction

`x[1] = 1003`. The remainder of `x` would be filled in similarly. In the example fragment, this was handled with a `for` loop.

To access the element associated with `x[2][1]` with this scheme, the compiler would have to find the third element of the 1-D array of pointers. It would find the address of this element by adding two times the space required for a `char *` to `x`. The value stored at this address should point to the start of the desired row. The compiler is looking for the second element of this row, so it adds the length of a single element (1 byte in this case) to this address, and voilá. Assuming that the `x` array was properly initiated, the row address retrieved in this hypothetical case would be 1006. The compiler would then add 1 to this address, getting 1007, and issue some kind of move instruction using this address to copy the element value wherever needed.

In case you are still confused, I can offer an analogy. Consider the following scheme for finding houses in a city. First, all the streets in the city are put into a numbered list in some set order.

1. Main
2. Elm
3. Adams
4. Broadway
5. State

3*dot*

Second, all the houses on each street are numbered in order, starting from zero. With that done, a particular house could be specified by giving two numbers, say [3][1]. The first number is the index in the street list of the street on which the house is located, and the second is the number of the house on that street. To find the house associated with a given address in this scheme, you would take the street index, go to the street list, find the street, go to that street, and then count houses to find the one with the specified number. Thus, the [3][1] house would be the second house on Broadway. (Remember, we start numbering the houses with 0.)

The analogy with the two-dimensional array scheme we have been discussing is the following. The street list is analogous to the array of pointers, `x`, in the example above. The individual streets are analogous to the 1-D arrays of data corresponding to the rows of the 2-D array, and the houses are analogous to the individual elements of these 1-D arrays. To use the scheme, you have to "build" the city by allocating memory for the houses, organizing them somehow onto "streets". You also have to create the "street list", and fill it in according to the "street organization" you have decided on.

What are the advantages and disadvantages of the two schemes? The principal advantage of the second scheme is that it is more general than the first. The actual data need not be stored in one contiguous block of memory. Each row of the 2-D array can be stored in a different location. Further, the rows need not have the same length. As you will see in an example, that feature can have advantages for storing character strings. The second scheme also has an advantage when the program involves a function to which arrays are passed as arguments. I'll discuss this point in the next section. Finally, the second scheme has the questionable advantage that it saves a little CPU time because accessing a particular element of the array requires only two additions, whereas the first also requires a multiply. The second scheme has the disadvantages that it requires a little more memory than the first (to hold the array of pointers), and it's a little more trouble to set up.

I'd like to discuss two examples of programs which use two-dimensional arrays. The first uses the first scheme for allocating the array, and the second uses the second scheme.

18.3.1 Adding Two Matrices

The first example program is shown in Fig. 18−5. It's a simple program that asks the user to type in two 3×3 matrices, adds them, and prints out the result. In this case, typing the two matrices into the computer is probably more trouble than just adding them by hand, but the program illustrates the use of two-dimensional arrays. The program uses the first method of creating 2-D arrays.

```
/*
 * This program adds two 3x3 matrices
 */

void main()
{
    float a[3][3], b[3][3], c[3][3], t;
    int i, j;

/*  The next statement squashes a bug in the Turbo C compiler.
 *  It is not needed with other compilers. */
    scanf("", &t);
/*  End of Turbo C bug fix. */

/*  Get the matrices */
    printf("FIRST MATRIX:\n");
    for(i=0; i<3; i++) {
        printf("Enter row No. %d:  ", i+1);
        for(j=0; j<3; j++)
            scanf("%g", &a[i][j]);
    }
    printf("SECOND MATRIX:\n");
    for(i=0; i<3; i++) {
        printf("Enter row No. %d:  ", i+1);
        for(j=0; j<3; j++)
            scanf("%g", &b[i][j]);
    }

/*  Add them */
    for(i=0; i<3; i++)
        for(j=0; j<3; j++)
            c[i][j] = a[i][j] + b[i][j];

/*  Report the results */
    printf("The sum is:\n");
    for(i=0; i<3; i++)
        printf("|%8g, %8g, %8g|\n", c[i][0], c[i][1], c[i][2]);
}
```

Figure 18−5. Example program which adds two 3×3 matrices and prints out the result.

Besides illustrating the use of two-dimensional arrays, the program also, unfortunately, illustrates a bug in the Turbo C compiler. Without the `scanf()` in the bug squasher section, the program bombs with a strange error message. Don't ask me why this fixes it. There is no such problem with other compilers I've tried the program on. `scanf()` can be tricky to use, but this is not supposed to be one of the problems! Besides stomping on the bug, the line has no other effect because `scanf` is given a format string without any specifiers in it. Thus, even though the address of the variable `t` is pushed onto the stack, `scanf` doesn't do anything with it because it doesn't know it's there. If you are using any compiler besides Turbo C, delete the line. The line won't hurt anything, but it's not needed.

The only other part of this program that needs further discussion is the use of nested `for` loops to read in the matrices and to do the addition. Consider the first set of `for` loops, just after the `printf`

statement printing "`FIRST MATRIX`." The body of the outer most loop (the one with index `i`) is executed three times, once for each row to be entered. In the loop the user is prompted to enter the proper row, and then the individual elements are read in, one at a time, using the second `for` loop (the one with index `j`). This `for` loop itself is executed three times (once for each value of `i`), and each time it is executed its body is executed three times. Thus the `scanf()` function gets called 9 times, once for each of the 9 elements of the matrix.

18.3.2 A Program for Averaging Test Scores

In this sub-section I'll consider a more involved example—a program intended to help an instructor average test grades in a class. There are three tests to be averaged for each student, and we assume there are no more than 30 students in the class. The program should read in each student's name, followed by the three test grades. It should then average the three grades for each student and print out the results.

This program is getting complicated enough that a preliminary program outline will be helpful. Fig. 18−6 shows the scheme I came up with.

```
/* Declarations */
        Declare names to be an array of character strings
        Declare scores to be a 2D array of scores.
          scores[i][j] gives the score of the ith student on
          the jth test.
        Declare avg to be an 1-D array to contain the averages

/* Get the names and scores */
        For(i=0; i< No. students; i++) {
              Get the next name
              Store it in name[i]
              Get the 3 scores for this student, store in
               scores[i][0], scores[i][1], and scores[i][2]
        }

/* Average the scores for each student */

        For(i=0; i< No. students; i++)
              avg[i] = average of 3 scores for this student

/* Print out the results */
        Print a heading
        For(i=0; i< No. students; i++)
              Print name[i] and average[i]

        Done
```

Figure 18−6. Program outline for a program to help an instructor average test scores.

The purpose of the outline is solely to help me write the C code for the program. The language used is one I invented as I went along, and the next time I write a program, I'll probably use a somewhat different one. As long as I understand what I mean, that's good enough. If this program were one that was going to be saved, and used by many others, it probably would be good to make some kind of program outline or flow chart with a more conventional format. Although there are a lot of details yet to be worked out, the outline shows the program structure I am thinking about it. As I start to write the C code, I'll probably discover some problems I hadn't anticipated that will necessitate some changes, but the outline gives me something

to work on.

Let's get on with writing the program. I have some decisions to make. First, `names` and `scores` are to be 2-D arrays. Which scheme should I use for creating them? I chose the second because `names` is to be an array of character strings, each of different and unknown length. With the first scheme, I would have to decide ápriori how much space to use for each name. If I erred and one name turned out to be longer than anticipated, the program would overwrite the space reserved for the next name. With the second scheme, I can write the program so that each name uses only as much space as it requires. I only have to allocate sufficient space for all the names. If one is longer than anticipated, that's OK as long as another is shorter than anticipated to compensate. As an aside, C has the capability of getting more memory while the program is running, so the program could be written so that I would not even have to guess how much total space to allocate for names. The computer would simply get more space every time it needed it. I will discuss that feature in the next chapter.

Another decision I have to make is how I will tell the computer how many students there are in the class. One way would be to just have the program ask, but I'm going to be trickier. I'll have the program simply start reading names and scores from the keyboard until it encounters a name consisting only of three Z's. That will be the signal that the last name has been read in already. The program itself will count the number of names entered before encountering ZZZ, and that will be the number of students.

How will I set up `names`, the array of pointers to the students' names? First, I'll allocate an array of characters large enough (I hope!) to hold all the names, and I'll set `names[0]` to point to the start of this block. I'll get the first name and store it, character by character, in this block, starting at `names[0]`. I'll then count the number of characters in this first name, including the terminating zero for the string, and set `names[1]` to `names[0]` plus this number. I'll then read in the second name, placing it in the space starting at `names[1]`, count the number of characters in this name, and set `names[2]` accordingly. I'll keep this up inside a `for` loop until I encounter the name of the terminator, ZZZ.

Fig. 18−7 shows the program I ended up with. There are several points which I should discuss about it. Fortunately, none of the points are Turbo C bugs this time.

First I'll discuss the setting up of the 2-D arrays. The memory space for holding the names is allocated in the first declaration statement,

```
char *names[30], name_mem[20*30];
```

The first part of the statement sets aside space for 30 pointers to be used for pointing to the start of the individual name strings. The second sets aside space for all the names. I've assumed that the names will on average require no more than 20 bytes each. The space for the scores is allocated in the second declaration statement,

```
int *scores[30], score_mem[3*30];
```

This sets aside memory to hold 30 pointers to sets of scores, and to hold 30 such sets of scores, each containing 3 scores.

These statements set aside the space needed, but they do not put the proper addresses in the arrays of pointers, `names[]` and `scores[]`. This task is started by the first two executable statements,

```
scores[0] = score_mem;
names[0] = name_mem;
```

The score sets and names will be stored one after the other in memory, with each occupying only as much space as needed. At this point we could continue assigning values to the remaining elements of `scores[]`, but we cannot do so with the `names` array because we don't yet know how long each name

```
/*
 *  Program to collect and average student's test scores for a class.
 */

void main()
{
    char *names[30], name_mem[20*30];    /* 30 names, 20 chars each */
    int *scores[30], score_mem[3*30];    /* 30 students, 3 scores each */
    char line[30], *p;
    int i, j, n_students, nchars, avg[30];
    float sum;
    char *gets(char *);                  /* Declares the gets function */

    scores[0] = score_mem;
    names[0] = name_mem;

/*
 *  The following for loop gets the names and the scores.
 */
    for(i=0; i<30; i++) {
    /* Get a name.  ZZZ means we're done */
        printf("Name (enter ZZZ when done):  ");
        gets(names[i]);
        if((names[i][0]=='Z') & (names[i][1]=='Z') & (names[i][2]=='Z'))
            break;

    /* Get three scores */
        printf("Enter three test scores.  ");
        gets(line);
        sscanf(line, "%d %d %d", &scores[i][0], &scores[i][1],
            &scores[i][2]);

/*  Set pointers for next loop iteration */
    /*  First figure out the length of name just entered */
        nchars = 0;
        for(p=names[i]; *p; p++)
            nchars++;

    /* The next name will go at the end of this one */
        names[i+1] = names[i] + nchars + 1;   /* + 1 for terminal 0 */

    /* Check that we have not run out of name space */
        if(names[i+1] > &name_mem[20*29]) {
            printf("Out of space for names\n");
            break;
        }

    /*  The next scores will go just after these three */
        scores[i+1] = scores[i] + 3;
    }

/*
 *  Now calculate the averages
 */
    n_students = i;
    for(i=0; i<n_students; i++) {
        sum = 0.;
        for(j=0; j<3; j++)
            sum += scores[i][j];

    /*  Round to nearest integer. Works only if sum >= 0.  */
        avg[i] = (sum + 1.5)/3.;
```

```
    }

/*  Print out the results */
    printf("The average scores are:\n");
    for(i=0; i<n_students; i++)
        printf("%20s  %3d\n", names[i], avg[i]);
}
```

Figure 18−7. An example program illustrating the use of two-dimensional arrays. It averages test scores, and prints out the student's names and test averages.

will be. We will have to fill in the elements of `names` as the names are collected, and it is most convenient to fill in the elements of `scores` at the same time.

The program then proceeds to get the names and scores. The name is read in using the statement

```
gets(names[i]);
```

The `gets()` function is a standard function supplied with the C compiler which reads a character string from the keyboard. It return a `char *` and has one argument, a `char *`, which is the memory location where the string is to be placed. I don't use the return value of `gets` in the program, but in case you are wondering, it returns the starting address of the string in memory (the same address as it was given as an argument). `gets()` detects the end of the string by the presence of a newline character (resulting from the user pressing **Enter**). The newline character is not returned. Instead the string is terminated in memory with a zero, in conformance with the C convention. In our case, we want the name to be placed into the block pointed to by `names[i]`, so that is the argument we supply to `gets()`. When `gets()` returns, the name will be in the desired place.

The program then needs to check to see if the name is the magic ZZZ. This is accomplished in the next two lines. The conditional of the `if` statement does the checking. The conditional will return true only if all three equalities are satisfied. The next line contains a new statement, `break`. This statement is executed only if the name ZZZ was entered, and causes whatever loop it is in to terminate. (If `break` is used inside nested loops, only the innermost loop is terminated.) In our program when the `break` is executed, the `for` loop terminates, and execution continues with the next statement after it: the one that reads `n_students = i;`.

If the awful name of ZZZ was not entered, the program prompts the user to enter the three scores, and then reads the response into the character array `line` using the `gets()` function again. This response is then decoded using `sscanf()`, and the results placed in the three elements of the `i`th row of the scores matrix. The function `sscanf()` is one we have not discussed before. It is very similar to `scanf()`, but it has an additional argument which is a pointer to a character string. Instead of reading the input, as `scanf()` does, `sscanf()` "reads" the string.

The first use of `gets()` seems pretty natural, but the second seems like a difficult way to go. Why not just use `scanf()` to read directly the input? The reason is that `scanf()` reads the input until it has satisfied all its format specifiers, and no farther. When we run the program we will enter the three scores and then press **Enter**. Pressing **Enter** tacks a newline character onto the end of the line, and if `scanf()` is used it will not read the newline because it doesn't need the newline to satisfy the three `%g`'s in its format string. The newline just stays there until the next iteration of the loop when we are trying to read the next name. A newline here is interpreted as the end of the name, and the very first character encountered will be the left-over newline. That will get things all messed up. Using `gets()` and then `sscanf()` has the advantage that it gets rid of the pesky newline.

The program needs to determine where the next name will be stored, and it now has enough information to do so. The next name should be stored just after the end of the one just read in, so the next address

is that of the current name increased by the length of the current name, including the terminating zero byte. The number of characters in the current name are counted and placed in the variable `nchars` using the instructions

```
nchars = 0;
for(p=names[i]; *p; p++)
    nchars++;
```

The pointer to the next name, `names[i+1]`, is then set. The program then makes a weak attempt to ensure that there is enough space for the next name, and breaks out of the loop if not. The check involves determining whether there are 20 or more bytes left in the memory allocated for the `name_mem` array. This check is far from foolproof, and this part of the program probably ought to be written better. The last statement in the loop puts the address for the (`i` + 1)th row of scores into `scores[i+1]`. We knew before running the program the length of each score array so this could have been done earlier, but it is convenient to do it here.

Once the student information is entered into the computer, the averages are calculated and printed out. There is only one curve ball here, on the line where the average is calculated,

```
avg[i] = (sum + 1.5)/3.;.
```

The right hand side of this expression is a `float`, and the left hand side is an `int`. When the quotient is stored in `avg[i]` it will be converted to an `int`, and the fractional part will be truncated. The reason for adding 1.5 to `sum` before dividing by 3 is to round the average rather than simply truncating it. This trick works as long as `sum` is positive, but fails if `sum` is negative.

18.3.3 Arrays as Function Arguments

How do you pass an array to a function? If the array has only one dimension, it's pretty simple. In fact we have already done so, in the programs in Figs. 18−1 and 18−2. One simply passes the function the address of the start of the array. Inside the function the corresponding argument is declared to be a pointer to an appropriate variable type. For example, in Fig. 18−1 the `str_no()` function is supposed to count the number of characters in a character string to be passed to it as an argument. The first line of the function definition was

```
int str_no(char *s)
```

and the function was called with the expression

```
str_no(line)
```

`line` being the name of a 1-D `char` array.

The situation is a bit more complicated when multidimensional arrays are used. To illustrate, let's modify the program in Fig. 18−5 so that it uses a function to add the two matrices. The result is shown in Fig. 18−8. The 2-D arrays corresponding to the matrices are set up using the first scheme. For simplicity, I chose to "hard-wire" in values for the two matrices to be added. This is done inside the first set of nested `for` loops, and the values I chose were arbitrary. Passing the arrays to the function was pretty simple. The arrays in the main program are declared as 3×3 arrays in the first line of `main`, so the arguments of `mat_add()` were declared similarly. The operation of the program should be self evident. Note the function prototype for `mat_add()` in the last line of the declarations near the start of the program.

A disadvantage of the `mat_add` routine in Fig. 18−5 is that it has the 3×3 dimensions of the arrays "hard wired" into it. If you want to add, say, 4×4 or even 4×3, arrays you have to rewrite the function and recompile. Why does the compiler need this dimensional information? It did not need such information when 1-D arrays were passed as arguments. The answer is that it only needs part of the information, and the part it needs is required in order that it be able to find the individual elements of the array. Consider what the compiler has to do to find the memory location holding, say, the element specified by `x[1][2]`. The three elements of the first row (index of 0) are stored sequentially, starting at the address given by `x`,

```
/*
 *  Program that uses a function to add two matrices.
 *  It illustrates how a multidimensional array can be passed
 *  to a function.
 */

void main()
{
        float a[3][3], b[3][3], c[3][3];
        int i, j;
        void mat_add(float [3][3], float [3][3], float [3][3]);

        /*  Initialize the matrices with arbitrary values */
        for(i=0; i<3; i++)
                for(j=0; j<3; j++) {
                        a[i][j] = 10*i + 4*j;
                        b[i][j] = 5*j - 7*i;
                }
        mat_add(a, b, c);
        printf("The sum is:\n");
        for(i=0; i<3; i++)
                printf("|%8g, %8g, %8g|\n", c[i][0], c[i][1], c[i][2]);
}

/*
 *  Function to add two matrices, x and y.  The result is returned
 *  in sum.  The matrices are assumed to have been allocated using
 *  the first scheme, and the matrices must have dimension 3x3.
 */

void mat_add(float x[3][3], float y[3][3], float sum[3][3])

{
        int i, j;

        for(i=0; i<3; i++)
                for(j=0; j<3; j++)
                        sum[i][j] = x[i][j] + y[i][j];
}
```

Figure 18−8. A program illustrating passing two dimensional arrays to a function as arguments. The program adds two 3×3 matrices.

then the three elements of the second row (index 1), and so forth. `x[1][2]` is the third element of the second row. To find this element, the compiler has to go the length of one row away from `x` to get to the start of the second row, and then to move three elements farther to find the actual element. More generally, to find the element specified by `x[i][j]`, the compiler would look in the address (`i*N_COLS + j`) times the length of an element, where `N_COLS` is the number of elements in a row. Thus, the compiler needs to know how many elements there are in a row, but it does not need to know how many rows there are. When the program runs, the function probably will have to know how many rows there are in order that it not try to access a row that doesn't exist, but the compiler does not need this information to create the machine language program.

The definition for `mat_add` in Fig. 18−9 would work similarly to that in Fig. 18−8, and it has the advantage that it can be used to add matrices with any number of rows. The matrices must still have three columns, however. There is an added argument, `nrows`, which is required so that the function will know how many rows there are, and the calling program will have to be modified to include this information in the declaration and in the calling statement for `mat_add`. The declaration would have to be changed to

```
/*
 *  Function that adds two matrices, x and y, and returns
 *  the result in sum.  The function assumes the arrays
 *  were allocated using the second scheme, and matrices
 *  with any number of rows may be added.
 *  nrows is the number of rows in the matrices.
 */

mat_add(float x[][3], float y[][3], float sum[][3], int nrows)

{
        int i, j;

        for(i=0; i<nrows; i++)
                for(j=0; j<3; j++)
                        sum[i][j] = x[i][j] + y[i][j];
}
```

Figure 18−9. A function definition which allows matrices with any number of rows to be added.

```
void mat_add(float [][3], float [][3], float [][3], int);
```

This version is more generally useful than its predecessor, but it still suffers from the restriction that the number of columns is "hard wired" into it. Is there a way to escape this problem? The answer is yes. The easiest way is to use the second scheme for allocating a 2-D array, and pass the function the address of the start of the 1-D array of pointers to the row arrays. We will also have to add an argument to the function to tell it how many columns there are. Fig. 18−10 shows the modified program and function. In the main program, the memory blocks labeled by `a_mem`, `b_mem`, and `c_mem` are where the actual matrix elements are stored, and the 1-D arrays of pointers, `a`, `b`, and `c`, contain the pointers to the individual row arrays inside these blocks. With this version, the `mat_add` function can be used without recompilation to add matrices of any dimensions. (There is still the requirement that the dimensions of the two matrices be added must be the same, but this is a requirement of the mathematics, not of the program.)

18.4 Preprocessor Directives and Stdio

Before the compiler sees your program, it is processed by another program called the *preprocessor*. Directives for this program consist of lines starting with a pound sign, #. I will discuss here two of the directives the preprocessor recognizes, `#define` and `#include`. There are a few more such directives, but I think its unlikely you will need them.

It's easier to show you how `#define`'s are used than to try to explain it explicitly. The program example in Fig. 18−7 would be a good place to use the `#define` feature. The program assumes that there are no more than 30 students. To change that requires making a number of changes to the program. Changes would be needed in many of the declaration statements, in a `for` statement, and in an `if` statement, and maybe elsewhere. It would make it much easier and more foolproof to make such a change if we had used a `#define` statement. Fig. 18−11 shows the first part of this program, rewritten in this way.

Every time the preprocessor sees the character string `N_STUDENTS` in the code, it replaces it with the character string `30`. Thus the compiler sees essentially the same program it did before, but the programmer sees a program which is considerably easier to interpret and change. To change the maximum number of students, one need only change the appropriate `#define` and recompile. Further, the use of a symbolic name makes the code easier to understand. As a general rule, I suggest avoiding "magic numbers". These are numbers which simply appear with no explanation what they are. It can be very hard to figure out where the numbers came from, or what they mean after you have written the program and put it away for a while.

The preprocessor will replace `#define`'d text anywhere it finds it in the program code except inside

```
/*
 *  Program that uses a function to add two matrices.  It
 *  illustrates passing a multidimensional array to a function.
 */

void main()
{
        float *a[3], *b[3], *c[3];
        float a_mem[9], b_mem[9], c_mem[9];
        int i, j, nrows, ncols;

    /* Initialize the pointer arrays */
        for(i=0; i<3; i++) {
                a[i] = &a_mem[3*i];
                b[i] = &b_mem[3*i];
                c[i] = &c_mem[3*i];
        }

    /*  Initialize the matrices with arbitrary values */
        for(i=0; i<3; i++)
                for(j=0; j<3; j++) {
                        a[i][j] = 10*i + 4*j;
                        b[i][j] = 5*j - 7*i;
                }

        mat_add(a, b, c, 3, 3);
        printf("The sum is:\n");
        for(i=0; i<3; i++)
                printf("|%8g, %8g, %8g|\n", c[i][0], c[i][1], c[i][2]);
}

void mat_add(float *x[], float *y[], float *sum[], int nrows, int ncols)

{
        int i, j;

        for(i=0; i<nrows; i++)
                for(j=0; j<ncols; j++)
                        sum[i][j] = x[i][j] + y[i][j];
}
```

Figure 18−10. Program illustrating the use of two-dimensional arrays as arguments to a function. The program adds two 3×3 matrices, but the function can be used to add matrices of arbitrary dimensions.

comments and between quotation signs. You should be careful that such text not appear in another context in your program because this instance will be replaced also. Thus, you should `#define` the string `if` to be something else only with the greatest care. It is common practice to use all capital letters in `#define`'s partly to avoid such a problem, and also to make it more clear what's a variable, and what's a constant which has been `#define`'d to look like a variable.

The `#define` statement can be used to define *macros*, but I won't go into that in these notes. If you are interested, consult any book on C.

The second preprocessor statement I want to discuss is the `#include` statement. The format is `#include "`*`filename`*`"`. This statement causes the preprocessor to find the file with name *`filename`*, and copy it into the program at the point where the `#include` statement is found. The file name can be specified in two ways. If it's placed between double quotes as above, the preprocessor takes the name to be just as written (between the quotes). If, on the other hand, it's placed between angle brackets, `<`

```
#define N_STUDENTS      30
#define NAME_SIZ        20
void main()
{
    int *scores[N_STUDENTS];
    int score_mem[3*N_STUDENTS];
    char *names[N_STUDENTS];
    char name_mem[NAME_SIZ*N_STUDENTS];
    char line[N_STUDENTS];
    char *p;
    int i, j, n_students, nchars;
    float sum;

    scores[0] = score_mem;
    names[0] = name_mem;

/*  Get the names and scores */
    for(i=0; i<N_STUDENTS; i++) {
        printf("Name (enter ZZZ when done):  ");
        gets(names[i]);
        if(names[i][0]=='Z' && names[i][1]=='Z' && names[i][2]=='Z')
            break;
        printf("Enter three test scores.  ");
        gets(line);
        sscanf(line, "%d %d %d", &scores[i][0], &scores[i][1],
            &scores[i][2]);

    /*  Set pointers for next loop iteration */
    /*  First figure out the length of name */
        nchars = 0;
        for(p=names[i]; *p; p++)
            nchars++;

    /*  Next names ptr points to character after the terminating 0
        of this name */
        names[i+1] = names[i] + nchars + 1;

    /* Check that we have not run out of name space */
        if(names[i+1] > &name_mem[NAME_SIZ*(N_STUDENTS-1)]) {
            printf("Out of space for names\n");
            break;
        }

    /*  Next scores ptr points to space just after the last 3 scores */
        scores[i+1] = scores[i] + 3;
    }
```

Figure 18−11. A modified version of the first part of the program in Fig. 18−7 which uses the `#define` preprocessor directive.

and > without the double quotes, the preprocessor will look in a special directory which contains the standard files which one might want to `#include`. For the Turbo C compiler on the lab computers, this directory is `\TC\INCLUDE`. I discussed one use of a file from this directory earlier in connection with declarations for functions in the standard math library. The `#include` directive is used frequently in C programs. These directives are usually placed near the beginning of the program file.

Probably the most common use of the `#include` directive is to make use of a more generalized input/output mechanism than what we've discussed so far. Suppose you wanted your program to write its output not to the terminal, but rather to a file. In that case, you would want to use the `stdio` package. Again, it's easier to show you than to tell you. The program in Fig. 18−12 calculates random numbers and

writes them out to a file named `rand.out`.

```
#include <stdio.h>
#include <math.h>

void main()
{
    int i, n, xrand;
    FILE *out;

    if((out=fopen("rand.out", "w")) == NULL) {
        printf("Can't open rand.out for output\n");
        exit(1);
    }
    printf("How many numbers?  ");
    scanf("%d", &n);
    for(i=1; i<=n; i++) {
        xrand = rand();
        fprintf(out, "%d\n", xrand);
    }
}
```

Figure 18−12. A simple program illustrating the use of `stdio` to write a set of random numbers to a file.

To use a file for I/O, you have to give it a name by which it will be known inside the program, and open it for either input or output. If you are using the `stdio` package (and I suggest you do), the internal name must be declared to be a pointer to a `FILE`. (The `FILE` type is not a basic C variable type, but it is defined in `<stdio.h>`. This is done using a `typedef` statement. I won't discuss this statements in these notes, but they are explained in any book on C.) I've chosen to call the file by the name `out`. To open a file, use the `fopen()` function. `fopen()` is declared to return a pointer to a `FILE` in `<stdio.h>`, and it has two arguments, both of which are `char *`'s. The first argument is the name of the file as the operating system knows it, and the second specifies whether you want to read or write (or both) the file. Use `"w"` for writing, `"r"` for reading, and see a C language manual for other possibilities. Using `"w"` creates the file if it does not exist, and truncates it to zero length if it does. If you use `"r"`, the file has to exist, or else `fopen()` bombs out. The function `fopen()` returns a pointer to the file opened, and the first executable statement of the program in Fig. 18−12 sets `out` equal to this pointer and tests the pointer to see if it's equal to `NULL`. `NULL` is `define`'d in `<stdio.h>`, and is the value that `fopen()` returns if it can't do what it was asked.

The program calls a function in the standard math library, `rand()`, which returns an integer, randomly distributed between `0` and `0x7fff`. The value returned is written to the file using the `fprintf()` statement. `fprintf` behaves identically to `printf()`, except that it has an additional argument which should be a `FILE *`, and it writes to the file this argument points to rather than the computer screen. For input, there is a corresponding function, `fscanf()`.

18.5 Type Casting of Variables

The C compiler is usually pretty good about figuring out automatically when variables should be converted from one type to another, but it doesn't always get it right. You can explicitly tell the compiler to convert a variable to another type by preceding the variable with the type you want it converted to placed in parentheses. That's called *casting* or *coercing* the variable from one type to the other.

Suppose, for example, that we had wanted to write out the square root of the random numbers in the previous example, rather than the random numbers themselves. That's easy, we would just use the `sqrt()` function of the math library to take the square root of the numbers returned by `rand`. There's

one small rub though, `sqrt()` takes a C`double` point argument, and `rand()` returns an `int`. The `int` must be converted to a `double`. One way to do that would be to declare `xrand` to be a `double` instead of an `int`. Then the statement `xrand = rand();` would cause the `int` returned by `rand()` to be converted to a `double`. You would also have to change the format specifier in the `fprintf()` call, of course. Another way would keep `xrand` declared as an `int`, but then explicitly cast it into a `double` when used as an argument as in the following line

```
fprintf(out, "%g\n", sqrt( (double) xrand));
```

There is still a third way to get the square root of `xrand`. Keep `xrand` declared as an `int`, as above, but use ANSI function prototypes to declare `sqrt()`. Then the compiler knows that the argument of `sqrt()` is supposed to be a `double`, and performs the conversion automatically. This method is particularly easy to implement because the `<math.h>` include file contains the ANSI prototypes of all the math function, so just

```
fprintf(out, "%g\n", sqrt(xrand));
```

would work with an ANSI compiler.

Explicit casting or coercion of variables is the point of this sub-section. You can explicitly cause the conversion of a variable from one data type to any other that makes sense in this way. As another example, in the following code fragment, the value in `fvar` is truncated and stored in `chopped`.

```
float fvar, chopped;

fvar = 3.14159;
chopped = (int) fvar;
printf("chopped = %g\n", chopped);
```

When executed, the computer would print the value `3`.

18.6 Exercises

1. Write a subroutine which can be used to calculate the value of a floating point number raised to an integer value. The subroutine should have two arguments; the first should be a `double` and contain the number to be raised to a power, and the second should be an `int` and contain that power. The function should return a `double` which is the value of the first argument raised to the second argument power. There is a standard math library function, `pow()` which could be used to implement the function, but you are not to use it. Your function should actually carry out the multiplications required. The function should work for integer powers which can be either positive, negative, or zero.

   Test your routine by using it with a test program which asks the user for the number to be raised to a power and the power itself, and then prints out the result. Your test program should prompt the user for what input is wanted, and it should make clear what's printed out. Choose numbers which completely test your subroutine, and turn in the results of your tests.

2. Write a subroutine which multiplies two matrices. Recall that if $\mathbf{A}$ and $\mathbf{B}$ are $m \times n$, and $n \times p$ matrices, then the product, $\mathbf{C}$, is an $m \times p$ matrix, the elements of which are given by

$$\mathbf{C}_{i,k} = \sum_{j=1}^{n} \mathbf{A}_{i,j} \mathbf{B}_{j,k}$$

   Your routine should have six arguments; the first two the matrices to be multiplied, the third the result, and the fourth, fifth, and sixth the dimensions of the two matrices to be multiplied. The function should not return a value. The matrices should all be arrays of `float`'s.

   Test your subroutine by writing a program which asks for the dimensions of the two matrices to be multiplied, inputs the two matrices from the keyboard, multiplies them, and then prints out the result in a convenient format.

3. Consider the following declaration statement.

```
int a[5][7];
```

   Discuss what the following expressions mean, and test your answer as well as you can by writing one or more simple C programs which print out the values of appropriate quantities. Part of the credit for this problem will be for your ingenuity in using the capabilities of C to demonstrate your answer.

   a. `a`

   b. `a[2]`

   c. `a[2][4]`

4. What do the following programs print out? In part b), specifically show blanks by using a lower case b for each blank on the line.

a.

```
#define MAX   5

main()
{
    int sum, i;

    sum = 0;
    for(i=1; i<=MAX; i++)
        if(i & 01)
            sum += i;
    printf("%d\n", sum);
}
```

b.

```
main()
{
    int i;
    float x;

    i = -6;
    x = 128.6;
    printf("i=%6d  i=%6x\n", i, i);      /* Careful! */
    printf("x=%12.3f\n", x);
}
```

c.

```
main()
{
    int i, *j;

    j = &i;
    i = 26;
    *j = 12;
    printf("i=%d\n", i);
}
```

d.

```
main()
{
    float *x, y[100];
    int i;

    x = &y[6];
    for(i=1; i<=4; i++)
        *x-- = (float) i;
    printf("%g\n", y[4]);
}
```

e.

```
main()
{
    int i;

    for(i=1; i<=10; i++)
        printf("%d, ", ++i);
}
```

5. Write a C program which writes the integers between 1 and 100 into a file named `junk.txt`. Each number should be on a separate line.

6. Write a program in C which inputs two integers from the keyboard and prints out the sum of all the integers between them, including the end points. For example, if you enter 2 and 5 into the program it should print out 14. You could either enter the smaller number first followed by the larger, or vice versa—2 then 5, or 5 then 2 for example. To get full credit on this problem, your program should work in either case.

7. Write a program which inputs a line of characters from the keyboard and prints out the number of characters in the line which are not spaces. For example, if you typed in the line

   C is easy!

   the program should print out 8.

8. What do the following simple C programs print out?

   a.

```
main()
{
    int n[100], i, sum;

    for(i=1; i<=5; i++)
        n[i] = i;
    sum = 0;
    for(i=1; i<=4; i++)
        sum += n[i];
    printf("%d\n", sum);
}
```

   b.

```
main()
{
    int n[100], *p, i, sum;

    p = &n[4];
    for(i=1; i<=5; i++)
        *p-- = i;
    for(i=0; i<=4; i++)
        printf("%d\n", n[i]);
}
```

c.

```
main()
{
    char *str, *w[10], *p;
    int i, j;

    str = "This is a string.";
    w[0] = str;
    i = 1;
    for(p=str; *p; p++)
        if(*p == ' ')
            w[i++] = p+1;
    for(j=0; j<i; j++)
        printf("%s\n", w[j]);
}
```

d. Assume the first `printf` prints `a = ff00`, and that an `int` occupies two bytes.

```
main()
{
    int a[10], i;

    for(i=0; i<10; i++)
        a[i] = i;
    printf("a = %x\n", a);
    printf("*a = %x\n", *a);
    printf("a[3] = %x\n", a[3]);
    printf("&a[3] = %x\n", &a[3]);
}
```

e. Assume the first `printf` prints `a = ff00`, and that an `int` occupies two bytes.

```
main()
{
    int a[5][3], i, j;

    for(j=0; j<3; j++)
        for(i=0; i<5; i++)
            a[i][j] = 16*i + j;
    printf("a = %x\n", a);
    printf("a[2] = %x\n", a[2]);
    printf("a[2][2] = %x\n", a[2][2]);
    printf("&a[2][2] = %x\n", &a[2][2]);
}
```

19. C PROGRAM EXAMPLES

In this chapter I will discuss three problems for which a C program provides a good solution. The first two are resurrections of problems we looked at previously, but used a spreadsheet to solve rather than a C program, and the third is new. These examples provide me the opportunity to show you some further features of C which can be very convenient. For the first two examples, which are similar to problems we solved earlier with a spreadsheet, notice the advantages and disadvantages of the two approaches. For simple programs, the spreadsheet is easier to set up, and its easier to make changes and to view the results. For more complicated programs, however, the spreadsheet becomes increasingly difficult and clumsy to program, whereas the added complications are usually handled much more easily with C. Finally, for long calculations especially, the C program executes much faster.

19.1 Simulating a Random Walk and Memory Allocation

The topic of this section will be the simulation of a random walk. In the process, I will also discuss *dynamic memory allocation*. This is a very nice feature of C which allows a program to get memory space from the operating system while the program is running. Thus, you don't have to guess beforehand how big to make arrays. Instead, you can write the program so that it gets space for the arrays while it is running, when it presumably knows how big the arrays are to be.

Recall that in Exercise 10−7 we considered the random walk problem. I suggest you review the problem before proceeding. Briefly, the problem is the following. Suppose a very drunk sailor is in the middle of a very narrow bridge with a high railing to keep him from falling off. He is so drunk that he's equally likely to take a step forward as backward. The question is, "on the average how far does he get from the center of the bridge after *N* steps?" As before, I'll use an RMS (Root Mean Square) average to calculate how far he gets, and I'll use the computer to simulate a large number of sailors, each on his own bridge. This time, however, I'll use C rather than Quattro Pro.

As usual, the first thing to do in writing a program is to decide on a general plan of attack. Here's what I came up with. The principal piece of information I want is the R.M.S. average distance travelled as a function of the number of steps taken. I'll simulate a large number of drunken sailors, using a random number generator to walk them, and I'll keep track of how far each has moved after each step. I'll walk the sailors one at a time, using a for loop. Inside the loop will be another loop in which I walk the sailor, one step at a time, using a random number generator. To calculate this average, I will need for each step the sum over all the sailors of the squares of the distances travelled after that step. For this purpose I'll use a 1-D array, `sum_2`, which has one element for each step number. As I am going through the sailors, I'll keep a running sum for each step number of the square of the distance that sailor has walked so far. Once I've gone through all the sailors, the rest is easy. I just have to divide each element in `sum_2` by the number of sailors, and take the square root of the result to obtain the R.M.S distance, averaged over all the sailors for each step number, and print out the results.

Fig. 19−1 shows a program outline. The program I finally wrote is shown in Fig. 19−2. It does the simulation, and writes the result out to a file named `sailor.out` in a format which makes it easy to import the result into Quattro Pro for graphing. The program generally follows the outline in Fig. 19−1, but I found it necessary to make several small changes. For simplicity, the program takes all steps to have the same length, and distances are reported in terms of this length.

This program first allocates enough memory space for `MAX_N_STEPS` steps. I decided it would be better to declare `sum_2[]` to be an array of `long`'s, rather than `int`'s because it will contain the sum of squares of distances, and this number could become larger than the ~32,000 maximum value of an `int`. (Actually, it probably would have been better to make it an array of `unsigned long`'s because it the values stored in it have to be positive.) The program then asks how many steps each sailor should take, and

```
Declare:
    int sum_2[MAX_N_STEPS]
    float rms_pos[MAX_N_STEPS]

Initialize sum_2 to zero

/* Loop over the sailors */
    For(sailor_no=0; sailor_no<n_sailors; sailor_no++) {
        position = 0

        /* Walk this sailor a total of nsteps steps */
        For(step_no=0; step_no<nsteps; step_no++) {
            x = random number in range -1 to +1
            If(x < 0)
                position--
            Else
                position++
            sum_2[step_no] += position2
        }
    }

/* Form R.M.S. averages from sums of squares */
    For(step_no=0; step_no<nsteps; step_no++)
        rms_pos[step_no] = sqrt(sum_2[step_no]/n_sailors)

    Print out all values of rms_pos
```

Figure 19−1. Outline of a program to simulate `n_sailors` drunken sailors trying to walk off a bridge.

how many sailors to simulate. If you ask for more steps than the program has space for, it sets the number of steps equal to the maximum it can handle. This is an annoyance for which dynamic memory allocation is a cure, and we will discuss that at the end of this section. The program then enters the main loop (`for(slr=1; ...)`) which sequentially walks each sailor through the requested number of steps. Each sailor is put at position zero, and then the `for(step=0; ...)` loop walks him.

The program uses the standard library function `rand()` to determine the direction of each step. Here I ran into another small problem. I had planned to generate a random number evenly distributed in the range -1 to +1, but `rand()` returns an integer between `0` and `7FFFH`. I could have converted the return value to a `float` distributed between -1 and +1, but instead, I changed my plan a little and step the sailor backwards if the return is in the lower half of `rand()`'s range, and forwards if it is in the upper half. The maximum value returned by `rand()` is `define`'d to be `RAND_MAX` in `<stdlib.h>`. Thus, I need only check to see if the returned value is less than `RAND_MAX/2`. The line `sum_2[step] += ...` calculates the sum of the squares of the positions of each sailor at each step number.

After the `for(slr=1; ... )` loop, the R.M.S. averages are caluculated. In doing so, it was necessary to call the `sqrt()` function with argument `sum_2[step]/nsailors`. There are two problems here. First, `sum_2[step]` is a `long` and `nsailors` an `int`, so the quotient will be truncated to a long integer. Second, `sqrt()` expects a `double` for an argument, but it will be given a `long`. Both problems can be solved by coercing one of the terms into a `double`. In the program, just to be explicit, I coerce both numerator and denominator, but only one is really necessary. The line then becomes

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define MAX_N_STEPS     1000

main()
{
    int nsteps, nsailors, step, slr, position;
    long sum_2[MAX_N_STEPS];
    float rms_pos[MAX_N_STEPS];
    FILE *out;

/*  Get the facts */
    printf("How many steps?  ");
    scanf("%d", &nsteps);
    printf("How many sailors?  ");
    scanf("%d", &nsailors);

/*  Protect against too large a value of nsteps */
    if(nsteps > MAX_N_STEPS)
        nsteps = MAX_N_STEPS;

/* Initialize sum */
    for(step=0; step<=nsteps; step++)
        sum_2[step] = 0;

    randomize();        /* Makes rand() more random */

/*  Loop over the individual sailors */
    for(slr=1; slr<=nsailors; slr++) {

  /*  Start the sailor at zero */
        position = 0;

  /*  Walk the sailor */
        for(step=1; step<=nsteps; step++) {
            if(rand() < RAND_MAX/2)
                position--;
            else
                position++;

  /*  Update the sum of squares */
            sum_2[step] += position*position;
        }
    }

/*  Form the RMS average */
    rms_pos[0] = 0.;
    for(step=1; step<=nsteps; step++)
        rms_pos[step] = sqrt(((double) sum_2[step])/((double)nsailors));

/*  Report the results */
    if((out=fopen("sailor.out", "w")) == NULL) {
        printf("Can't open sailor.out for output\n");
        exit(1);
    }
    for(step=1; step<=nsteps; step++)
        fprintf(out, "%d,%g\n", step, rms_pos[step]);
}
```

Figure 19−2. C program to simulate an arbitrary number of drunken sailors each on a narrow bridge, and to calculate the RMS average distance the sailors walk measured from the starting point.

```
rms_pos[step] = sqrt(((double) sum_2[step])/((double) nsailors));
```

Notice that its necessary to tell the compiler that `sqrt()` is a function returning a `double`. That was done automatically for me when I `include`'d the header file `<math.h>`. Without the `#include` statement, I would have required a declaration statement `double sqrt(double);`.

Finally, the program opens the `sailor.out` file, and writes the results to it. I chose a format which would make it easy to import the data into Quattro Pro for graphing (use the **Only commas** option).

19.2 Dynamic Memory Allocation

The program in Fig. 19−2 works. I'd like to take this opportunity to show you a very nice feature that C has—dynamic memory allocation. In the random walk program in Fig. 19−2, we had to guess how much memory to allocate to the arrays `sum_2` and `rms_pos`. If we need less memory, then that's no problem, although we will waste memory space, but if we need more space, we'll have to edit the program and recompile. C has a function called `malloc()` which gets memory and returns a pointer to the start of the memory block it fetched. The argument of the function is the number of bytes requested. We can use this function in our program to allocate the required amount of memory automatically.

How much memory do we need to ask for? For `sum_2` we need room for `nsteps` `long`'s, and for `rms_pos` we need room for the same number of `float`'s. How much room do each of these variables require? We could look that up in the Turbo C reference manual, or write a test program like those in Chapter 17 of the notes. Easier is to use the `sizeof` operator. For example, we could use either `sizeof(long)` or `sizeof(*sum_2)` for the number of bytes required for one element of `sum_2`. The first part of the program with these changes is shown in Fig. 19−3. In the program the memory `malloc()` returns is supposed to contain in one case a set of `long`'s, and in the other a set of `float`'s. Notice that for this reason the return value of `malloc()` is coerced to be a pointer to the proper type of variable before it is assigned. This is mostly an academic point because pointers to all types of variables are stored in exactly the same way, so the coercion is trivial.

19.3 Using C to Solve Differential Equations

In Chapter 13 we discussed numerical methods for solving differential equations, and implemented them using a spreadsheet. You can use C for the same purpose, and in this section, I'll use C to solve the differential equation which results from analyzing the circuit shown in Fig. 19−4. The circuit is similar to that in Fig. 13−3, except that I have added a component we haven't talked about, called a diode, in parallel with the resistor, capacitor, and inductor. A diode is a device which to some approximation allows current to flow through it in only one direction, the direction indicated by the arrow in the schematic diagram symbol. A pretty good approximation to the voltage-current characteristic of a diode is given by the following equation,

$$i_D(v) = I_0(e^{v/v_T} - 1) \tag{1}$$

where I_0 and v_T are constants. I_0 depends on how the diode is constructed and has value typically in the range 10^{-12} to 10^{-10} Amp. v_T depends on the temperature and at room temperature is about 0.0257 volts.

Proceeding as in Section 13−3, we get the same initial conditions,

$$v(t) = 0$$

and

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <alloc.h>

main()
{
    int nsteps, nsailors, step, slr, position;
    long *sum_2;
    float *rms_pos;
    FILE *out;

/*  Get the facts */
    printf("How many steps?  ");
    scanf("%d", &nsteps);
    printf("How many sailors?  ");
    scanf("%d", &nsailors);

/*  Allocate memory */
    if((sum_2=(long *)malloc((nsteps+1)*sizeof(long))) == NULL) {
        printf("Can't get memory for sum_2\n");
        exit(1);
    }
    if((rms_pos=(float *)malloc((nsteps+1)*sizeof(float))) == NULL) {
        printf("Can't get memory for rms_pos\n");
        exit(1);
    }

/* Initialize sum */
    for(step=0; step<=nsteps; step++)
        sum_2[step] = 0;

...
```

Figure 19−3. Segment of the modified version of the program in Fig. 19−1, which uses dynamic memory allocation for the arrays `sum_2` and `rms_pos`.

$$i_L = \frac{V}{R_S}$$

For times after the switch is opened, applying the no net current rule as in 13−3 gives, similarly to Eq. 13−36,

$$C\frac{dv}{dt} + \frac{v}{R} + i_L + i_D(v) = 0 \qquad (2)$$

where we use Eq. (1) for $i_D(v)$. This equation differs from Eq. 13−36 only in the addition of the diode current term. This term converts the differential equation from a linear equation which can be solved pretty easily to a strongly non-linear equation for which an analytic solution is not known (by me at least). This equation is, therefore, a realistic candidate for numerical solution.

I choose to proceed as in Section 13−3, using the modified Euler's method to solve the coupled set of two first order differential equations in v and i_L. Everything goes as in 13−3 except for the additional term from the diode. Eqs. 13−40 become

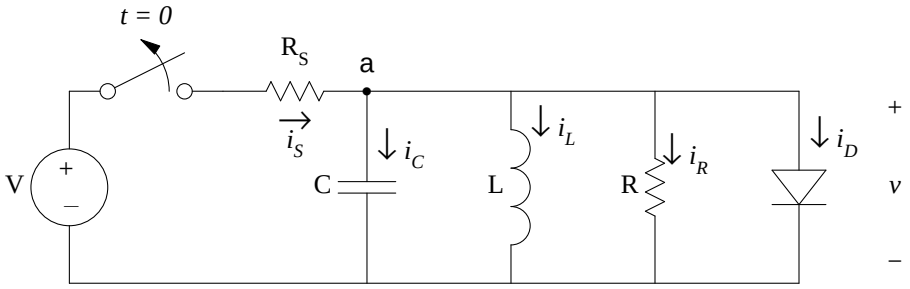

Figure 19−4. Schematic diagram for the circuit to be analyzed.

$$i_{k+1} = i_k + \frac{v_{k+\frac{1}{2}}}{L}\Delta t$$

$$v_{k+1} = v_k - \left(\frac{i_{k+\frac{1}{2}}}{C} + \frac{v_{k+\frac{1}{2}}}{RC} + \frac{i_D(v_{k+\frac{1}{2}})}{C}\right)\Delta t$$

and Eqs. 13−41 become

$$i_{k+\frac{1}{2}} = i_k + \frac{v_k}{L}\frac{\Delta t}{2}$$

$$v_{k+\frac{1}{2}} = v_k - \left(\frac{i_k}{C} + \frac{v_k}{RC} + \frac{i_D(v_k)}{C}\right)\frac{\Delta t}{2}$$

Writing a program which implements this algorithm in C is pretty straight forward, so I won't show you a program outline. The C program is shown in Fig. 19−5. As in the previous section, I chose to print out the values to a file in a format which would make it easy to import the data into a Quattro Pro spreadsheet for plotting. I ran the program with the following values for the parameters:

$L = 0.01$ H,	$C = 10^{-6}$ F,	$R = 10^4\ \Omega$,
$I_0 = 10^{-11}$ Amp,	$V_0 = 2$ Volt,	$R_s = 100\ \Omega$,
$N = 1000$,	$\Delta t = 2$ x 10^{-6} Sec.	

A plot of the result for v is shown in Fig. 19−6.

One of the first things you should do after you obtain the results of such a calculation is to look at them critically to see if they make sense. The most striking thing about these results is the asymmetry—notice the location of the zero on the voltage axis. Starting from $t = 0$, the voltage decreases smoothly to about −2 volts, and then starts increasing, apparently heading toward something like +2 volts, but the rise is abruptly terminated at just above 0.5 volts. After getting through this region, the waveform continues on to make a relatively symmetric sine wave. Does this behavior make sense?

What's causing the flattening of the first positive peak? Is it an effect of the diode current term, or did I make a mistake in my program? Rerunning the program with I_0 set to zero gives a pretty sine wave like I got for the circuit in Fig. 13−3. Therefore most of the program is probably OK. The flattening must be caused by the diode term, but I still don't know if it should "really" be there, or if I made a mistake in the diode current part of the program, or if there's something strange going on with the numerical method.

Could the diode itself be responsible? Fig. 19−7 shows the I-V curve for the diode with the parameters used in the program. The current for reverse bias (negative voltage) is negligible. I chose to plot the voltage range between 0 and 0.6 volts because that's the range where the funny stuff happens in Fig. 19−6. From what the program wrote out to the output file, `rlcd.out`, the typical inductor current is in the range of 10 mA. That current must go somewhere. For the value of R used, the current through the resistor is

```
/*
 *  Program to calculate the response of a parallel RLC circuit
 *  with a diode connected across the circuit.
 */

#include <stdio.h>
#include <math.h>

#define FILENAME        "rlcd.out"
#define VT     0.0257
float I0;

main()
{
    float L, C, R, dt, ihalf, vhalf, dt2, V0, Rs, RC;
    float *v, *i, *t;
    float iD(float);
    int npts, k;
    FILE *out;

    if((out=fopen(FILENAME, "w")) == NULL) {
        printf("Can't open %s for output\n", FILENAME);
        exit(1);
    }

/*
 *  Get the facts:
 *                  R, L, and C are the component values,
 *                  I0 is a diode parameter and should be of the
 *                       order of 1e-11 to 1e-12,
 *                  V0 is the voltage source value, and Rs the
 *                       series resistance
 */
    printf("L, C, R, and I0?  ");
    scanf("%g %g %g %g", &L, &C, &R, &I0);
    printf("V0, Rs?  ");
    scanf("%g %g", &V0, &Rs);
    printf("Number of points?  ");
    scanf("%d", &npts);
    printf("dt?  ");
    scanf("%g", &dt);

/*  Get memory */
    if((v=(float *)malloc(npts*sizeof(float))) == NULL ||
        (i=(float *)malloc(npts*sizeof(float))) == NULL ||
        (t=(float *)malloc(npts*sizeof(float))) == NULL) {
            printf("Can't get memory\n");
            exit(1);
    }

/*  Set initial conditions */
    i[0] = V0/Rs;
    v[0] = 0.;
    t[0] = 0.;

/*  Define two constants */
    RC = R*C;
    dt2 = 0.5*dt;

/*  Calculate i and v */
    for(k=1; k<npts; k++) {
        ihalf = i[k-1] + v[k-1]*dt2/L;
```

```
            vhalf = v[k-1] - (v[k-1]/RC + i[k-1]/C + iD(v[k-1])/C)*dt2;
            i[k] = i[k-1] + vhalf*dt/L;
            v[k] = v[k-1] - (vhalf*RC + ihalf/C + iD(vhalf)/C)*dt;
            t[k] = t[k-1] + dt;
        }

    /*  Write it out to the output file */
        for(k=0; k<npts; k++)
            fprintf(out, "%g,%g,%g\n", t[k], v[k], i[k]);
    }

    float iD(float v)

    {
        return(I0*(exp(v/VT) - 1.));
    }
```

Figure 19−5. C program to calculate the time dependence of v and i_L in Fig. 19−1. The program writes the results out to a file so that they may be graphed conveniently using a spreadsheet.

insignificant, so the current must go through either the capacitor or the diode. From the diode I-V curve, the diode current is negligible on this scale as long as v is less than about 0.4 volts, so all of the inductor current must flow onto the capacitor, charging it up. For these voltages, the diode might as well not be there, and the circuit should behave very similarly to that in Fig. 13−3 of the notes. When the voltage gets near about 0.5 volts, however, the diode current becomes significant, and some of the current that would have gone into charging the capacitor gets diverted through the diode. The rate of charging of the capacitor decreases, therefore, until at the clamped peak nearly all the inductor current is flowing through the diode. Finally, the voltage starts to fall as the capacitor starts to discharge first through the diode, and then through the inductor. From this point on, the voltage stays below about 0.5 volts, and the diode no longer plays an important role.

This voltage limiting behavior of diodes is commonly seen in circuits. Because of the shape of the I-V curve, the diode does not allow the voltage across it to rise above some "knee" value which is typically in the range 0.5−0.7 volts, depending on the diode and on the scale of currents in the circuit. A common approximation in circuit analysis is that $v_D \approx 0.6$ V whenever significant current is flowing through the diode in the forward direction. You will make use of that approximation frequently in analyzing and designing circuits with bipolar transistors in them.

For voltages less than this "knee" value, the diode current is small, and the diode is effectively out of the circuit. The diode current remains small for negative voltages up to much larger magnitudes. For sufficiently large and negative voltages, the diode breaks down, and the current increases sharply. As long as the current is limited to a reasonable value, this breakdown does not harm the diode, and the reverse breakdown characteristic can also be used to clamp a voltage at a fixed value. Diodes specially made to have a well-controlled reverse breakdown characteristic are called *Zener diodes*, and are used to provide a reference voltage for regulated power supplies and other uses.

19.4 Recursive Function Calls

C allows a function to call itself. A function which does that is said to be *recursive*. In this section I will give two examples of recursive functions. The first calculates the factorial of a positive integer, and the second calculates the determinant of a square matrix of arbitrary dimensions. Both functions use recursion in a natural way.

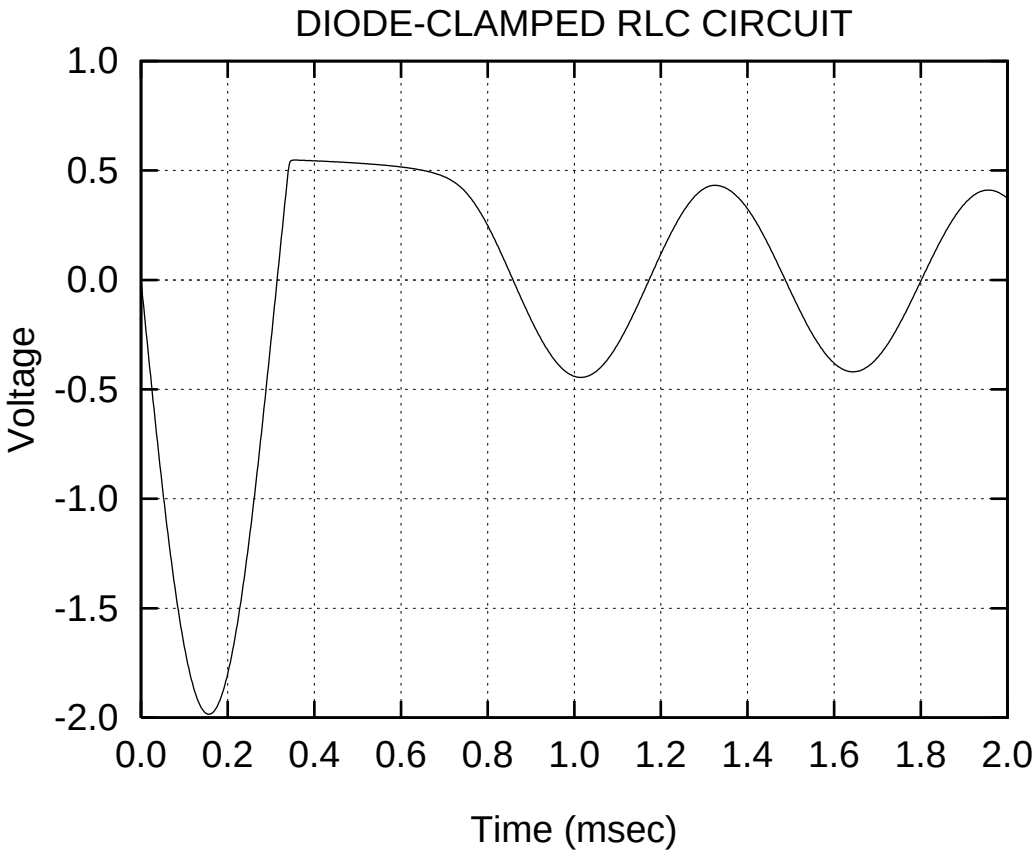

Figure 19−6. Calculated voltage vs. time for the circuit in Fig. 19−1.

19.4.1 n Factorial

In section 15−7.3 we wrote a recursive subroutine for the PFW007 to calculate the factorial of a number. In this subsection, I will translate this program into C, and add a few refinements. I suggest you review that section now to remind yourself how the algorithm works. Briefly, the idea is to make direct use of the definition of the factorial of a positive integer or zero:

$$n! = n(n-1)!$$

$$0! = 1$$

The function will have one integer argument, which is the number to be "factorialed," and will return a `double` which is the factorial of the argument. I chose to have it return a `double` instead of an `int` because the factorial of even modest numbers can become quite large.

The program is shown in Fig. 19−8. I think it's pretty straight-forward. The subroutine `fact()` checks to see if `n` is zero. If yes, it returns `1.`, otherwise it calls itself with an argument of `n-1`, and returns `n` times the result.

Several improvements are possible. First, `fact()` really should check `n` to see if it's negative and take some appropriate action if so. If one gives the existing function a negative argument, it will never return. (Well it actually will return, first because `n` is stored as a 16-bit integer and if you keep decrementing it it will eventually become zero; and second because the product in the return statement will after not too long become larger than even a `double` can handle, and the program will abort with an overflow error.

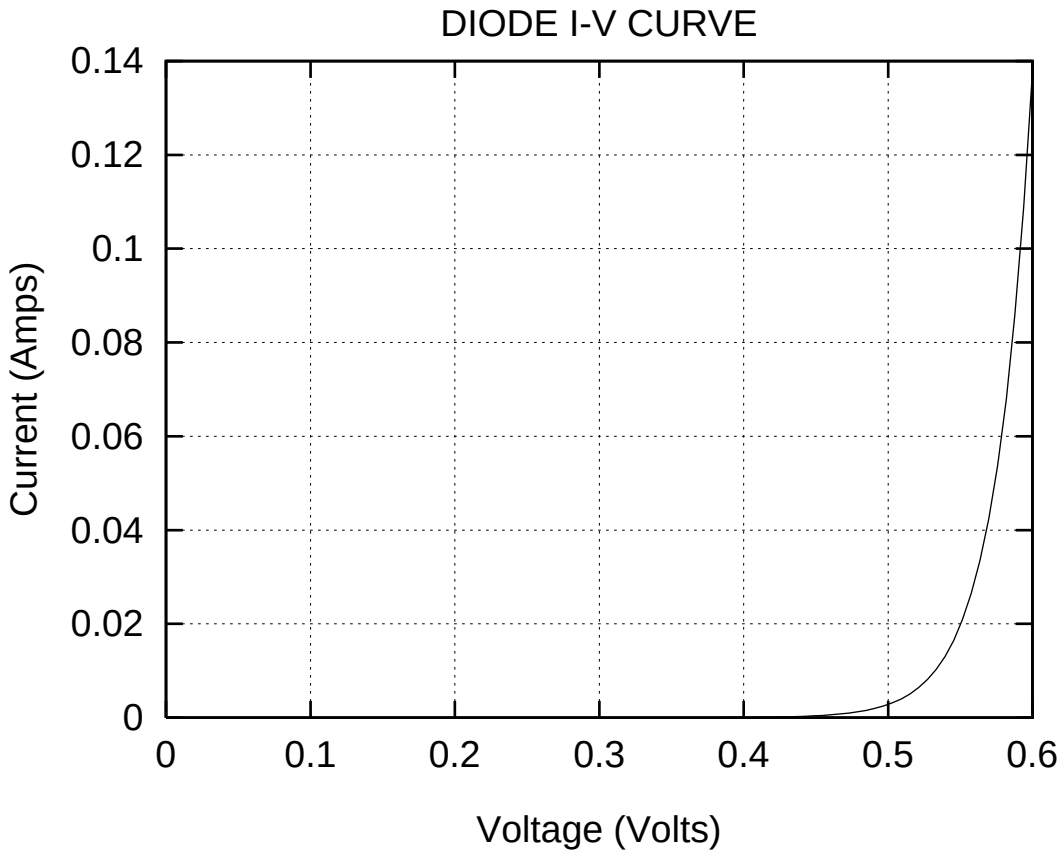

Figure 19−7. Current-voltage relationship for the diode used in the circuit described by Fig. 19−4.

In any case, the result is probably not what the user had in mind!)

This is not a very good way to calculate factorials because each function call takes time. The function could be speeded up considerably by going in steps of 2 or three, or more each time through. For example, instead or returning $n(n-1)!$, the function could return $n(n-1)(n-2)[(n-3)!]$. That would cut the number of function calls required by a factor of 3. The routine would have to check on input whether or not `n` was 0, 1, or 2 in order that it never overshoot into negative numbers. If `n` were 0, 1, or 2, the function would return 1, 1, or 2 respectively. I won't take this example any farther, but you should be able to implement these improvements easily if you want to.

19.4.2 Determinant

There is little justification for using a recursive function for the purpose of calculating a factorial because the non-recursive algorithm using simply a `for` loop is faster, and no more complicated or difficult to understand. I'd now like to consider another example of recursion which is better justified. This routine calculates the determinant of a square matrix of arbitrary dimensions. It turns out that there are better (faster, less susceptible to round-off errors) methods, but this one is the easiest to understand. I can easily explain to you how this one works, but I'd have a much more difficult time with the other methods.

The method works the same way you probably learned to calculate determinants—it expands by minors. Briefly, to calculate the determinant of an $n \times n$ matrix by this method, choose a row or column of the matrix and draw a line through it. Say you chose a column. Then for each element of this column draw a line through the row that intersects with this element and calculate the determinant of the resulting $(n-1) \times (n-1)$ matrix. Doing that yields n determinants, one for each element of the crossed-out column.

```
/*
 *  Simple program to test the recursive factorial calculator.
 */

void main()
{
    int n;
    double fact(int);

    printf("What's n?  ");
    scanf("%d", &n);
    printf("%d! = %g\n", n, fact(n));
}

/*
 *  Recursive subroutine to calculate the factorial of an integer.
 *  It works according to the definition of a factorial:
 *       n! = n * (n-1)!
 *       0! = 1
 */
double fact(int n)

{
    if(n == 0)
        return 1.;
    return n*fact(n-1);
}
```

Figure 19−8. Simple subroutine which uses recursion to calculate the factorial of a positive integer, and a main program for testing it.

Call the i^{th} of these n determinants $D_i^{(n-1)}$. Also call the i^{th} element of the crossed out column v_i, then the desired determinant of the original matrix is

$$\text{Det} = \pm \sum_{i=1}^{n} (-1)^{i-1} v_i D_i^{(n-1)} \tag{19−1}$$

where the + sign is used if the chosen column number is odd, and the − sign is used if even. The determinants of each of the $(n-1)\times(n-1)$ matrices are then calculated in the same way until the problem has finally been reduced to lots of 2×2 matrices, which are calculated as

$$\begin{vmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{vmatrix} = a_{1,1}a_{2,2} - a_{2,1}a_{1,2} \tag{19−2}$$

For example, the determinant of the following 3×3 matrix would be calculated with this method using the right-most column as follows.

$$\begin{vmatrix} 1 & 3 & 1 \\ -2 & 1 & 2 \\ -1 & 2 & 1 \end{vmatrix} = 1\cdot\begin{vmatrix} -2 & 1 \\ -1 & 2 \end{vmatrix} - 2\cdot\begin{vmatrix} 1 & 3 \\ -1 & 2 \end{vmatrix} + 1\cdot\begin{vmatrix} 1 & 3 \\ -2 & 1 \end{vmatrix}$$

$$= 1\cdot(-3) - 2\cdot(5) + 1\cdot(7)$$

$$= -6$$

Fig. 19−9 shows a recursive C function which uses this algorithm, along with a main program to test it.

Let's look at the main program first. For purposes of verifying how the function works, I've written the function so that it writes out its arguments to a file each time it is called. That's the reason for the statement opening the debug file. The declaration for `dbug` is placed outside of `main()` so that `det()` knows about it. Notice the three statements used to allocate space for the matrix. First `x` is set equal to the address of a block of memory large enough to hold `n` pointers to `float`'s. Each of these pointers is then set equal to the address of individual blocks of memory large enough to hold `n float`'s. The function `getmem()` is a home-brew subroutine defined at the end of the program file which simply calls `malloc()` and checks the return for error. After memory for the matrix is allocated, values for the elements in it are entered. Finally, all the work is done in the last `printf()` statement.

Let's look now at the `det()` function. The function has two arguments, one a pointer to a pointer to a `float` which tells it where to find the two-dimensional matrix for which the determinant is needed, and the other is an `int` which gives the dimension of the matrix. First, for debugging, it prints out to a file the matrix it was passed as an argument. Once we are convinced it is working properly, this feature should be removed from the program. It then checks the dimension argument, and returns the value of the determinant directly if it is 1 or 2. Otherwise, it expands by minors. It turns out to be easiest to program if we expand along the right-most column, as in the example above. It forms the n minors (each with dimension $(n-1)\times(n-1)$), and calls itself with each of these matrices. Once these calls have returned, the value of the determinant is calculated according to Eq. (19−1), and returned.

There is a small complication here because we must return `sum` if the column number about which we are expanding is odd, and `-sum` if it is even. Since we are expanding about the right-most column, the question is whether `n` is odd or even. The statement `if(n & 01)` checks this. The conditional is true if the lowest bit in `n` is set, and false otherwise. If the lowest bit is set, `n` is odd; if not, `n` is even.

Each iteration of the main loop, `for(i=0; ...)`, calculates the determinant of one of the `n` minors. Each minor is a matrix made up of the same elements as the given matrix, except that the right-most column and one row are missing. Memory is allocated for an array of `n-1` pointers to `float`'s which will contain the address of the `n-1` rows in the minor, and these pointers are set appropriately, omitting the i^{th} row, in the following two loops, `for(j=0; j<i; ...)` and `for(j=i+1; j<n; ...)`. The function is then called again with this matrix, and the return value used to calculate the contribution of this minor to the overall determinant. These contributions must alternately add to and subtract from the overall result, and the variable `sign` is used to keep track of that. Finally, the statement `free(p);` returns the memory used for the array of pointers back to the computer's heap to be allocated on some other occasion. Without this statement, if a number of determinants were to be calculated, the computer could run out of memory.

I have several comments about this program. First, a non-recursive version of it can be written which would execute faster. The non-recursive version would have a considerably more complex structure and would be more difficult to write. The only difficult part of the recursive program is the stuff needed to form the minors, and this would still be needed in the non-recursive version. Second, the allocation of two-dimensional matrices as an array of pointers to arrays of `float`'s made this section of the code easier to program, and faster to run. For each minor, instead of having to allocate space for and copy all $(n-1)\cdot(n-1)$ elements of the matrix to form the minor, we need only deal with space for and copying of a single array of $(n-1)$ elements. That's a savings of a factor of $(n-1)$. Since n could be 10 or larger, that's significant.

Finally, as an aside, this is a workable technique for calculating determinants, but it is neither the fastest nor the most accurate. The recommended method involves calculating two matrices such that the original matrix is the product of the two, and such that one of the new matrices has only zeroes above the diagonal and the other only zeroes below. The determinant of the original matrix is the product of the

```
/*
 *  Program to test the recursive determinant calculating
 *  subroutine.  The program includes a debugging feature
 *  in which the minors are written to a file as they are
 *  calculated.
 */

#include <alloc.h>
#include <stdio.h>

FILE *dbug;       /* Note the external declaration of dbug */
                  /* so det() will know about it.          */

void main()
{
    int n, i, j;
    float **x, z;
    float **getmem(int), det(float **, int);

/*  Open the debug file */
    if((dbug=fopen("debug","w")) == NULL) {
        printf("Can't open debug for output\n");
        exit(1);
    }

/*  Get the facts */
    printf("Size of matrix?  ");
    scanf("%d", &n);

  /*  Work around the Turbo C bug */
    scanf("", &z);

  /*  Allocate memory for the matrix */
    x = getmem(n*sizeof(float *));
    for(i=0; i<n; i++)
        x[i] = (float *)getmem(n*sizeof(float));

  /*  Get the matrix */
    printf("Enter matrix\n");
    for(i=0; i<n; i++) {
        printf("Enter row No. %d:  ", i+1);
        for(j=0; j<n; j++)
            scanf("%g", &x[i][j]);
    }

/*  Calculate and print out the determinant */
    printf("Determinant is %g\n", det(x, n));
}

/*
 *  Recursive determinant calculator subroutine.
 *  It calculates the determinant by expanding by minors.
 *  For debugging, the routine writes out the matrix it is
 *  passed each time it's called to the debug file.
 */

float det(float **x, int n)

{
    int i, j, sign;
    float **p, **getmem(int), sum, det(float **, int);
```

```
/*  Print out the matrix for debugging */
    fprintf(dbug, "det(x, %d):\n", n);
    for(i=0; i<n; i++) {
        fprintf(dbug, "     |");
        for(j=0; j<n; j++)
            fprintf(dbug, "%10.4g,", x[i][j]);
        fprintf(dbug, "|\n");
    }

/*  If it's a 1x1 or 2x2 calculate the determinant directly and return */
    if(n == 1)
        return(x[0][0]);
    if(n == 2)
        return(x[0][0]*x[1][1] - x[0][1]*x[1][0]);

/*  Otherwise expand by minors */
    sum = 0.;
    sign = 1;
    for(i=0; i<n; i++) {
        p = getmem((n-1)*sizeof(float *));

      /*  Omit the ith row */
        for(j=0; j<i; j++)
            p[j] = x[j];
        for(j=i+1; j<n; j++)
            p[j-1] = x[j];

        if(sign > 0)
            sum += x[i][n-1]*det(p, n-1);
        else
            sum -= x[i][n-1]*det(p, n-1);
        sign = -sign;
        free(p);
    }
    if(n&01)         /* if n is odd */
        return sum;
    else        /* if n is even */
        return -sum;
}

/*  Subroutine to get nbytes of memory.  It's just malloc()
 *  with error checking.
 */

float **getmem(int nbytes)
{
    float **p;

    if((p=(float **)malloc(nbytes)) == NULL) {
        printf("Can't get memory\n");
        exit(1);
    }
    return p;
}
```

Figure 19−9. Recursive subroutine to calculate the determinant of a matrix of arbitrary dimension, along with a main program to test it.

determinants of the two factor matrices, and, because of their special form, the determinant of the factor matrices is just the product of the diagonal elements. Once the factor matrices are found, calculating the

determinant is an easy matter. Actually, it's even easier than that because you can always arrange things so that one of the factor matrices has only ones on the diagonal so its determinant is simply 1. This is all interesting stuff, but I can't go into it here. The factoring of the original matrix into two pieces is called *LU decomposition*. LU decomposition is also a good way to invert a matrix.

19.5 Exercises

1. Copy the random walk program in Fig. 19−2 with the modifications in Fig. 19−3 into a file, compile, and run it. Plot your numerical results using Quattro Pro. According to the discussion in Exercise 10−7 of the notes, after N steps, the average sailor should have gotten $L_{step}\sqrt{N}$ away from the center of the bridge. Plot this prediction also on the same graph and compare the two. Turn in a floppy with both your C program and this worksheet on it.

2. The random walk program in the chapter assumed the sailors always took steps of the same length—only the direction was random. That's not very realistic. Modify the program so that each step has a random length varying between 0 and 1. Run the program and plot the result as in the previous exercise. For a constant length step, you should have found that the average sailor did in fact move a distance about $L_{step}\sqrt{N}$ from the bridge center. What would you guess should replace this prediction in this case in which the step size varies randomly? The answer is given in the next problem. Does your simulation produce this result?

3. In the previous exercise, you were to write a C program to simulate numerically an "average" drunken sailor on a bridge. The sailor to be simulated was to take steps with length varying randomly between 0 and 1. After n steps, it can be shown analytically that on the average, the sailor will have moved a distance of $\langle L_{step}\rangle\sqrt{n}$, where if one uses an R.M.S. average of positions, $\langle L_{step}\rangle$ is the R.M.S. average step length. Show analytically that for this case $\langle L_{step}\rangle = \sqrt{\frac{1}{3}}$.

   HINTS:

   > Remember what R.M.S. means—the square root of the average of the squares of the quantity to be averaged.
   >
   > To form the average, you first have to add the squares of the lengths of all the steps. Imagine dividing the interval $[0, 1]$ up into a large number of equal-width subintervals, each of width say Δx. If you are considering a large number of steps, say N, how many of the steps will have length falling in one of these subintervals? If the length associated with the i^{th} subinterval is x_i, about how much does the i^{th} subinterval contribute to the sum?
   >
   > In the limit that Δx becomes very small, does the resulting sum look anything like something we discussed in Chapter 11?

4. In Chapter 13 we discussed Euler's method for solving first order differential equations numerically. The method involved using a forward-time difference approximation to the derivative, i.e.

$$\left.\frac{dy}{dt}\right|_i \approx \frac{y_{i+1} - y_i}{\Delta t}$$

   In this problem, you are to derive a similar formula based on using the backward-time difference approximation to the derivative,

$$\left.\frac{dy}{dt}\right|_i \approx \frac{y_i - y_{i-1}}{\Delta t}$$

   This method is sometimes called the *implicit* Euler's method. Apply the method to the following differential equation,

$$\frac{dy}{dt} = -\frac{y}{\tau}$$

   subject to the initial condition $y(0) = 1$. Here τ is a constant which would be provided at the time

the equation was to be actually solved. Write a C language program which solves this equation using the implicit method for $\tau = 1$. Test your program and turn in a floppy containing the program and a Quattro Pro worksheet containing a graph of the difference between your numerical result and the correct analytic result. From the dependence of the error on step size, what is the order of the method?

Implicit methods are frequently used to solve some types of differential equations numerically because they tend to be more stable than explicit methods. This stability often comes at the price of more complicated arithmetic operations, however.

5. In designing oscillators, the following equation, called the *van der Pol* equation sometimes arises.

$$\frac{d^2v}{dt^2} - G(1 - v^2)\frac{dv}{dt} + v = 0$$

where v is related to the voltage in the circuit, and G is related to the gain of the amplifier in the circuit. This is a dreaded non-linear differential equation, the analytic solution of which is very difficult. The numerical solution is much easier, however.

Write a C program which uses the modified Euler's method to solve the differential equation. The program should ask the user for the value of G, the initial values of v and dv/dt, the time step to be used, and the number of time points for which v is to be calculated. The program should write the results out to a file in a format suitable for plotting them using Quattro Pro.

Try running your program for several values of the initial conditions and parameters, and see if the solutions look anything like what you might expect from an oscillator. Try whatever values you find interesting, but turn in a floppy with worksheets containing graphs of $v(t)$ for at least the following two sets of parameters.

v_0	$(dv/dt)_0$	G	Δt	No. Pts.
1×10^{-3}	0	0.3	0.2	500
10	0	0.3	0.05	500