

Learn to Program In GSoft BASIC

By Mike Westerfield

Copyright 1999

Byte Works[®], Inc.

8000 Wagon Mound Dr. NW
Albuquerque, NM 87120-2845

Voice (505) 898-8183

FAX (505) 898-4092

E-Mail MikeW50@AOL.COM

Web <http://www.hypermall.com/byteworks>

Table of Contents

Lesson One — Getting Started	7
Before We Get Started.....	7
How to Learn to Program	9
What You Need.....	9
What You Should Already Know.....	10
GSoft BASIC, The FREE Version!.....	11
Getting Everything Ready.....	11
The Three Faces of GSoft BASIC.....	11
Your First Flight... er, Program.....	12
Dealing with Errors	15
A Close Look at Hello World	16
More About Reserved Words.....	16
Case Sensitivity	17
Where Are The Line Numbers?	18
How Programs Execute	19
Graphics Programs	20
Lesson Two — Variables and Loops	25
Integer Variables	25
More About Variable Names	27
Using DIM To Declare a Variable Type	29
The FOR Loop	30
Some Thoughts on Comments	31
Operator Precedence.....	32
The Maximum Integer	34
Real Numbers.....	35
PRINT USING for Dollar Amounts.....	38
Exponents.....	40
Why So Many Kinds of Numbers?	41
Lesson Three – Input, Loops and Conditions	43
Input.....	43
Our First Game... er, Computer Aided Simulation	45
The DO-LOOP	46
The Flexible DO-LOOP Statement	49
Random Numbers.....	50
Why Random Numbers Are Important.....	52
The IF Statement	54
The ELSE Clause	56

The World's Shortest Animation Course.....	57
Nesting If Statements.....	60
A Bit of Iffy History	62
Boolean Logic	63
Lesson Four – Subroutines.....	67
Subroutines Avoid Repetition	67
The Structure of a Subroutine	70
Where to Put Subroutines	72
The END Statement.....	72
Commenting Subroutines	72
Procedure Description.....	73
Parameters	73
Shared Variables.....	74
Return Values	74
Notes	74
Subroutines Let You Create New Commands	74
Functions are Subroutines that Return a Value.....	77
Value and Variable Parameters	80
Shared Variables	85
Lesson Five –Strings.....	89
What Are Strings?	89
The Two Ways To Read a String	89
Manipulating Strings	91
Characters	96
The ASCII Character Set	97
The Extended Character Set.....	98
P-Strings, C-Strings, and Other Confusions	100
Comparing Strings.....	101
Numbers and Strings	102
Garbage Collection.....	102
Lesson Six –Arrays.....	105
Groups of Numbers as Arrays.....	105
The Shell Sort.....	109
Multidimensional Arrays	115
Passing Arrays to a Subroutine	123
Lesson Seven – Types and Constants.....	129
Simple Types and Named Types.....	129
The Six Built-in Types.....	129

The TYPE Statement	131
CONST	131
Records Store More than One Type	132
Lesson Eight – Files	135
An Overview of the Process	135
Opening a File for Output	136
Writing to a File	137
Closing a File	137
Writing Our First File	138
Reading from a File	138
File Names, Directories, Path Names and Folders	140
File Names, GS/OS and ProDOS	140
Path Names	141
Partial Path Names and the Default Prefix	142
Names in Programs	142
Colons and Slashes	143
Finding the End of a File	143
Printing with Files	146
Binary Files	147
Opening and Closing Binary Files	147
Writing Binary Files	148
Reading Binary Files	149
Reading and Writing Practically Any File	150
More About File Types and File Formats	150
Random Access	151
Lesson Nine – Pointers and Lists	157
What is a Pointer?	157
Pointers are Variables, Too!	158
Allocating and Deallocating Memory	160
How New and Dispose Work	161
Linked Lists	163
Stacks	164
Queues	169
Running Out Of Memory	171
Lesson Ten – Miscellaneous Useful Stuff	173
The SELECT CASE Statement	173
Revisiting the FOR Loop	178
The GOTO Statement	182

The ONERR GOTO Statement	183
Variant Records	185
A Quick Tour of Some Advanced GSoft BASIC Features	196
Changing the Size of Memory	197
Libraries	197
The MakeRuntime Utility	198
Lesson Eleven – Scanning Text.....	199
The Course of the Course	199
Manipulating Text	200
Building a Simple Scanner.....	201
Symbol Tables.....	205
Parsing	207
Lesson Twelve – Recursion	213
A Quick Look at Recursion	213
How Procedures Call Themselves.....	213
Recursion is a Way of Thinking.....	215
A Practical Application of Recursion.....	218
Lesson Thirteen – Sorts.....	225
Sorting.....	225
The Shell Sort.....	225
Quick Sort	228
How Fast Are They?.....	234
Quick Sort Can Fail!.....	235
Sorting Summary.....	236
Lesson Fourteen – Searches and Trees.....	237
Storing and Accessing Information.....	237
Sequential Searches	237
The Binary Search.....	238
A Cross Reference Program for BASIC.....	239
The Binary Tree	248
Ruffles and Flourishes	256
Index	259

Lesson One — Getting Started

Before We Get Started...

When I went to grade school, my teachers tried to beat some basic skills into my thick head. Back then, the basic skills included reading, writing, and arithmetic. When it came to spelling, my mind was already warped, because my teachers had also explained that these were the three R's.

Lately, in our rapidly changing world, we have added a new basic skill. It just isn't good enough to be able to read and write, plus do some math. In 1965, it wasn't easy to get from New York to Chicago without reading signs, writing instructions, counting some change and reading a clock. Today, you will use a computer to make the same trip. The travel agent will log your reservations in a computer. You may get spending money from a computer-based automatic teller. A digital watch counts bits to tell you what time it is. Computers control the flow of trains and the displays used by air traffic controllers. Your check book may even have a calculator. It's become a computerized world, and people who can't or won't deal with computers are rapidly being left as far behind as an illiterate person in the sixties.

Of course, you know all of that. That's why you have decided to learn to program. By the end of the course, you will know one of today's most popular and widely available programming languages, BASIC. You will know it well enough to write programs of your own. Whether you want to plot an engineering equation, create a custom cooking program to adjust ingredients for any number of people, or write a computer game, this course will get you ready.

If you have been around computers for a long time, you may know that there are many languages you can use to write programs for your computer. It's fair to ask why you should learn BASIC.

One of the things you must look for in a computer language is that it must be fairly common. If a language is common, that tells you two important things: A lot of people think the language is a good one, and no matter what computer you decide to write a program for, you are likely to find the language you know. Today, there are five languages that fulfill this first requirement. They are C, Pascal, assembly language and BASIC.

If you decide to make your living programming a computer, you will eventually learn most of these languages, and probably a few more. If you are learning to program, though, you have to pick just one of them to learn first. We can immediately rule out

assembly language. In assembly language, you have to deal with the machine's internal structure. It takes many individual instructions to do the simplest thing. You will spend more time dealing with bits and bytes than learning how to write a well-organized program

C and Pascal are both good choices. Compared to BASIC, most people find C rather obtuse. The reason has to do with the type of programming each language was created for. One of the design goals of the BASIC language was to create a simple language for scientists and engineers. BASIC has grown beyond this initial audience, but it is still one of the simplest of the popular computer languages to learn and read. It has all of the facilities needed to implement modern programs. C was designed for professional programmers who implement programs that might need to do some very tricky things. Because of its built-in safety checks, the BASIC language often hinders their efforts. C, on the other hand, does not have these checks. That's good for the careful professional programmer, but bad for a beginner, who really needs those checks

Pascal is a great first language, just like BASIC. Pascal tends to be more verbose, but that's not necessarily a bad thing, because Pascal programs tend to be laid out better as a result. We'll learn some techniques for laying out BASIC programs to get around this minor disadvantage.

BASIC has an advantage over both C and Pascal, though. There are two common ways to create computer languages. One method is called compiling, and the other is called interpreting. At this point, it's not important to understand the technical difference between the two, just to realize there are two methods, each with its own advantages and disadvantages.

Most implementations of BASIC, including GSoft BASIC, are interpreters. That's good and bad. Interpreted programs run slower than compiled programs, although that's not an important consideration for any of the programs in this course, or for many other programs. But interpreted programming languages are easier to use. Your program doesn't have to be compiled, so it is ready to run right away. There are also fewer steps in creating a program, which makes your job of learning to use the language a lot easier.

I'd like to make one point clear, though. BASIC is not inherently slower than Pascal or C. A BASIC compiler will create programs that run at about the same speed as compiled Pascal or C programs. For that matter, you could create an interpreter for Pascal or C, and in fact, that has been done. Any language can be implemented either way, but BASIC is generally implemented as an interpreter, and Pascal and C are generally implemented as compilers.

Before getting too much further, I also want to point out what this course is not. This is not a course about writing Apple II GS desktop programs. I don't want to discourage you from writing desktop programs; quite the contrary. On the other hand, as you will find out, there is a lot to learn about programming before you are really ready to tackle

something like a desktop program. By the time you finish this course, you will be ready to start to learn about desktop programming. If you tried to learn desktop programming right away, though, you would probably fail. There's just too much to learn to try and do it all at once.

How to Learn to Program

Learning to program has a lot in common with learning to fly an airplane. When you learn to fly, most people start with an introductory flight with an instructor. Those that don't often make a bad first landing, and never get a second chance. (An old adage around flight schools is that any landing you walk away from was a good landing.) Before, during and after the flight, the instructor will tell you about some of the basics of flight: How the control surfaces work, what the controls do, and so forth. There will be a lot you don't know, and a lot of things you are told may not make sense right away. As you progress, you will spend time reading books and sitting in lectures, but you will also spend a lot of time actually flying the airplane. You wouldn't expect to spend all of your time reading books and sitting in lectures, then walk out to the plane and go off for a cross-country flight with no instructor; you gradually work up to that point. Eventually, though, you solo. You start to fly long distances, first with an instructor and then alone. Finally the day comes when you get your license.

It's the same way with programming. In a moment, we'll get started. We'll start off with a few simple programs. It is absolutely essential that you type them in and run them. There will be many problems that you can work on your own. The more problems you work, the better programmer you will become. Sure, we will spend some time talking about the ideas behind programming, and there will be some problems that you need to work through with a pencil and paper. For the most part, though, you will be programming; either typing in and analyzing programs with the help of this material, or writing and running your own programs. Gradually, the programs will get longer, and before long you will be able to write your own programs.

Just in case you missed the point, let me spell it out in very simple terms. If you read this material, but don't type in the sample programs or work the problems, you will know as much about programming as you would know about flying from reading a book. In short, very little. Programming is a skill. If you don't practice the skill, you will never learn it.

What You Need

Now is the time to sit down in front of your computer. Before starting, let's make sure you have everything you will need. First, you need an Apple IIGS computer. (An emulator is fine, as long as it emulates an Apple IIGS.) It must have a monitor. It's nice if the

monitor is color, and you'll use color in some graphics programs, but you can make do with a black and white monitor. The computer must have at least 1.125M of memory. For the older Apple IIGs that came with 256K on the mother board, this means that the memory card in the special memory slot must be populated with 1M of memory. In the most common case of an Apple memory card, this means that there should be a memory chip in each socket on the card. You can check this by taking the top off of your computer and looking. With the newer Apple IIGs, which comes with 1.125M of memory on the mother board, you don't need a memory card at all.

You must have at least one 3.5" disk drive, plus another 3.5" disk drive or a hard drive. It is possible to use GSoft BASIC with a single 3.5" disk drive, but it's tough. We won't go over the details of using a single drive here. If you don't have a second drive, contact Byte Works technical support and discuss your options.

You will need a copy of GSoft BASIC. If you decide to use a different BASIC, there will be some things in this book that will not work. You would have to figure out why and make appropriate adjustments. By the time you finish this course, you will know enough to do that. At first, though, you may not. For that reason, I would suggest that you stick with GSoft BASIC.

There are some other things that would be nice, but not essential. Most people like to print their programs and look at the paper copy. I highly recommend a printer if you intend to try this. A hard disk is also very nice. Hard disks can hold much more than a floppy disk, so you will not have to switch disks as often. Hard disks are also faster than floppy disks, which again speeds up the programming process. Finally, an accelerator card will roughly double the speed of your computer. As I said, all of these are nice. If you end up spending a lot of time programming, I would encourage developing a close relationship with St. Nicholas in an attempt to collect these items. You can, however, do everything in this course without them.

What You Should Already Know

Any book about computers has to make some assumptions about what you already know. Let's briefly discuss the assumptions I'm making so you're not surprised about things I leave out.

I assume you're a reasonably intelligent person who is already familiar with using computers. I won't be telling you how to insert floppy disks, how to use an editor to type programs, or how to copy files. For the most part, you should already know how to do those things. Using the text editor is the one area where you may need a little more help than usual, simply because the editor we'll use to write programs is a little different than editors used to write books and letters. The similarities are more frequent than the differences, though, and a little time with the GSoft BASIC reference manual should be enough for anyone who already knows how to use a word processor.

GSoft BASIC, The FREE Version!

There are two versions of GSoft BASIC. This book assumes you are using the commercial version, but almost everything in this book will actually work on the smaller, free version. You can download a copy of the free version from <http://www.hypermall.com/byteworks>. You can also get a copy on a floppy disk from the publisher of this book for a small fee.

If you're not sure which version is for you, start with the free one. When you get to the point in this course where it deals with libraries and creating programs that run from the Finder, which are the two things the commercial version adds that we use in this course, you can switch to the commercial version of GSoft BASIC or just skip those sections.

Getting Everything Ready

When I bought my first FORTRAN compiler for the Apple II, I had a frightening experience. I wrote a program that crashed the compiler. The program actually erased some of the information on the compiler disk, so I could not use that disk anymore. In those days, many vendors still took the absurd position that computer languages had to be copy protected. My local dealer either could not or would not help me restore the disk. I had one other copy (the program came with two copies), but I was afraid to use it.

Fortunately, times have changed. Computer languages are no longer copy protected. The very first thing you should do when you open your copy of GSoft BASIC is to make copies of both of the floppy disks that come with the package. You can use the Finder to do this. If you know how to use some other copy program, and you like it better, go ahead and use it. Any copy program will work. Label each of the disks you have copied, and put the originals in a safe place.

If you are using two 3.5" floppy disk drives, you will use a copy of the first disk in the second floppy disk drive and your normal boot disk in the first drive. You might want to make an extra copy of the first GSoft BASIC disk; you'll use that one both to run GSoft BASIC and to save your programs.

If you are using a hard drive, you will need to install GSoft BASIC on the hard drive. You can find instructions in the reference manual that comes with GSoft BASIC.

The Three Faces of GSoft BASIC

It's worth mentioning that there are three ways to create and run a GSoft BASIC program. This course assumes you are using the simplest, the GSoft BASIC shell. (A shell is the name for the program that looks at things you type, like CATALOG or EDIT, and carries out your instructions.)

There is another version that runs from the ORCA shell. All of the programs you see here will work fine from that version. If you're already familiar with the ORCA shell, you have the commercial version of GSoft BASIC, and you prefer the ORCA shell to the smaller, simpler one built into GSoft BASIC, feel free to use it. If you have any trouble getting started with the ORCA shell version, refer to the reference manual.

The last version of GSoft BASIC works in conjunction with either of the first two. Whether you are using the GSoft BASIC shell or the ORCA shell, the programs you create only run from that environment. Of course, you will eventually create a program you want to give to other people, and they may not have GSoft BASIC. Even if they do, they will want to run the program like any other program, from Apple's Finder or some other program launcher. The third version of GSoft BASIC is a special one that lets you create programs that will run from the Finder. We'll cover how to create these programs, and discuss the advantages and disadvantages, later in this course.

Your First Flight... er, Program

It's time to take that first test flight. Strap yourself in. After all, as you have no doubt heard, computers can crash, so always wear your seat belt. Fortunately, though, a computer crash caused by programs you write in this course won't hurt anything.

As we go through this program, there will be a lot you don't understand. Be patient; in time, you will. The one thing you should keep in mind, though, is that you can't write a program that will damage the computer. Even if you do something wrong, the absolute worst thing that will happen is you will erase a disk—and even that is so unlikely that it isn't worth worrying about very much. It is, however, worth worrying about enough to make a copy of the GSoft BASIC disks, which is why you should never run from the original disks. You should also keep backups of your hard drives. Frankly, it's more likely that you will lose information on your hard drive from the hard drive wearing out than from the programs in this course, but it's always best to keep good backups, just in case.

From Apple's Finder, locate the file GSoft.Sys16, either on your hard disk or on the second 3.5" floppy disk. Double-click on GSoft.Sys16 to start the program. You'll see a banner across the top of the screen with the program's version number, a blank line, and a } character followed by a cursor. From here, you can type any of the commands you find in



Chapter 5 of the GSoft BASIC reference manual. We won't use many of the commands in this course, but we will cover each of them we use in detail. Some of the commands we don't cover may come in handy, though, so plan to flip through Chapter 5 of the GSoft BASIC reference manual at some point, just to see what's there.

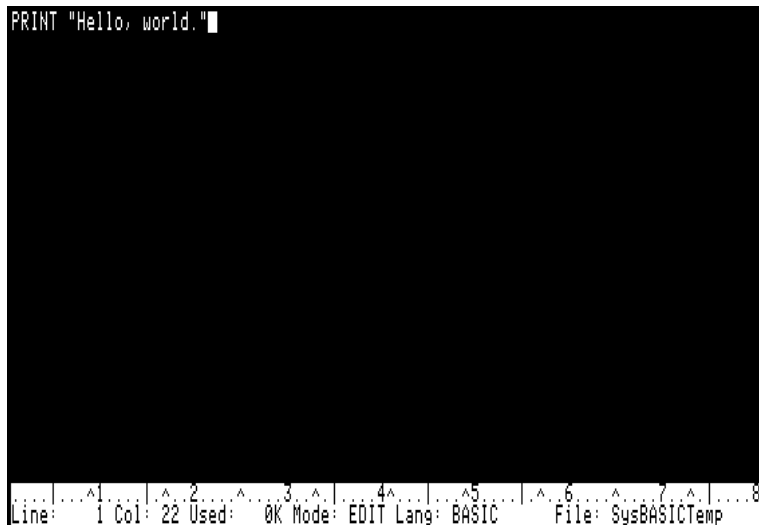
When you write a program, you type the program pretty much the same way you would type a letter in a word processor. Our first step, then, is to enter the editor so we can type a program. From the GSoft BASIC shell, type the line

```
EDIT
```

and press the return key. It doesn't really matter whether you use uppercase or lowercase letters.

The edit command shifts you from the shell to the full screen editor. The editor that comes with GSoft BASIC is optimized for writing programs, so it's a little different from editors like AppleWorks that are intended for writing letters, but there are also many similarities. We'll assume that you have used enough editors that you can type the program and use cursor keys and the return key to move around the screen. If you need help, try Command-?, which brings up the editor's help screen, or refer to Chapter 6 of the GSoft BASIC reference manual.

Type in the following program. The format isn't terribly critical, but since you don't know what is and what is not important yet, it is best to type it exactly as shown. This program writes the characters "Hello, world." to the screen. It's a simple program, but we must start somewhere.

A screenshot of the GSoft BASIC editor. The main window is black with white text. The first line of code is "PRINT 'Hello, world.'" followed by a cursor. At the bottom, there is a status bar with a grid of column numbers (1-8) and the following text: "Line: 1 Col: 22 Used: 0K Mode: EDIT Lang: BASIC File: SysBASICTemp".

```
PRINT "Hello, world."
Line: 1 Col: 22 Used: 0K Mode: EDIT Lang: BASIC File: SysBASICTemp
```

```
PRINT "Hello, world."
```

The next step is to exit the editor. To be honest, this is a little peculiar. We'll discuss the reasons in a minute. To exit the editor, type Command-Q. The editor will ask you if

you want to save the program. Select yes, and you're dropped back into the GSoft BASIC shell.

At this point, you have created a BASIC program that is stored in the memory of your computer. GSoft BASIC calls this the program buffer, or sometimes the workspace. If you type

```
LIST
```

You'll see the program itself. You can also look at or change the program using the editor, typing

```
EDIT
```

again. In fact, technically there is a program in the program buffer as soon as you start GSoft BASIC—it just doesn't have any lines. It's legal to run a completely empty program, it just doesn't do anything.

It's a good idea to get in the habit of saving your program to disk occasionally. Right now, if you leave GSoft BASIC, your program vanishes. That may not be a big deal for a one-line program, but you'll want to save your longer ones.

To save the program to disk, type SAVE followed by the name you want to use for the program. Save this program as Hello, like this:

```
SAVE Hello
```

This command saves your program to the same folder where GSoft.Sys16 is located. There is nothing to stop you from putting the programs somewhere else, but your life will be a little easier if you just leave the programs in the same folder as GSoft.Sys16. Occasionally, you will want to copy the programs to a separate archive disk and delete them from your working folder, especially if you are using floppy disks. You can actually do this from GSoft BASIC's shell, but this is one area where the Finder's desktop interface works better. Personally, I use GSoft BASIC's file manipulation commands if I need to copy or move one or two files, but for large numbers of files, I leave GSoft BASIC and use the Finder.

With the program safely saved to disk, just in case a catastrophe happens, it's time to spin the prop and see if it flies. Type

```
RUN
```

This command executes the program in the program buffer. It's the GSoft BASIC equivalent of double-clicking on a program from the Finder. This program consists of a simple PRINT statement that writes the text between the quote marks to the screen, so you'll see

```
Hello, world.
```

on your display. Congratulations, you've just written and executed a computer program!

I mentioned earlier that the way the editor worked was a little peculiar. After all, the editor saved the program as you left the editor to go back to the GSoft BASIC shell, right? Well, sort of. Without getting too deep into technical explanations, the editor ended up saving the program in the workspace, not on the disk. Maybe you heard the disk being accessed. The editor and GSoft BASIC are actually separate programs. They use the disk to pass the program back and forth, but the temporary file they use for communication is erased automatically.

Dealing with Errors

If your program didn't quite work, check each step to see what you did wrong. One of the most common programmer mistakes is to assume that any mistake is the computer's fault. Sorry, it just ain't so. If things didn't work it's because, in order of likelihood:

1. You didn't do exactly what you were told.
2. You don't have the correct hardware.
3. You have a bad disk.
4. You have bad hardware.

Just for the record, you should know that absolutely every program we will show you in this course has been mechanically moved from the word processor to GSoft BASIC (or vice versa) and executed. They have also been typed in and executed by one or more guinea pigs we call beta testers, who we use to find errors before the errors can confuse you. In other words, every single program has been tested at least twice. If you encounter a problem, the chances are very, very small that the problem is in the program in the course material or in GSoft BASIC.

Now, that's not to say GSoft BASIC is perfect. You may encounter a bug someday. Still, the overwhelming number of programmers, especially beginning programmers, who blame a problem on the machine or the programming language have made two mistakes: One in the program itself, and the other in assigning the blame to the wrong source! The

fact is, programming can be very humbling, because the mistakes are almost always your own.

To correct a problem, go back over each step in the text. If there is an error message, examine it closely to see what it tells you. The GSoft BASIC reference manual has an appendix that lists all of the errors and some common causes. Check your typing in the area very carefully; a typing error is the most common cause of errors at this stage.

A Close Look at Hello World

Now that you have actually run a program, let's stop and spend some time talking about what happened. We'll start by examining the program in detail. The first step is to take a look at the words that make up the program

Like sentences in a book, programs are made up of a series of words and punctuation marks. Some of the words have special meaning, while some are words we pick to name parts of the program.

We'll dissect our first program to look at some of these rules. This program consists of a single line. All BASIC programs are organized as zero or more lines. Most of the time, each line contains a separate, distinct command for the computer to carry out. The entire collection of lines is called the program.

```
PRINT "Hello, world."
```

The word PRINT is called a reserved word. This just means that you can only use the word PRINT in special ways in a BASIC program. It is also a statement in BASIC; it's one of the commands the language understands. The characters that we want the program to write are placed after the word PRINT. BASIC uses the quote character to mark the start and end of a string constant. This lets you write things like parenthesis, reserved words, and so forth, without confusing BASIC. As long as you keep the string on one line, you can put absolutely any characters you want in the string, except for the quote mark itself. You can still write a quote mark, but it takes a little more work. We'll look at how it's done in Lesson 5.

More About Reserved Words

In the last section I pointed out that our first program had something called a reserved word, and that reserved words can only be used in special ways. PRINT is one of the reserved words in GSoft BASIC. Here's a complete list of the reserved words:

ABS	ALLOCATE	ALLOCATEPROC	AND	APPEND
AS	ASC	AT	ATN	BINARY
BREAK	BYTE	CALL	CASE	CDBL
CHDIR	CHR\$	CINT	CLEAR	CLNG
CLOSE	CONST	CONT	COS	CSRLIN
CSNG	CURDIR\$	DATA	DEF	DIM
DIR\$	DISPOSE	DISPOSEPROC	DO	DOUBLE
ELSE	END	EOF	ERL	ERR
ERROR	EXP	FN	FOR	FRE
FUNCTION	GET	GOSUB	GOTO	GSOS
HCOLOR=	HEX\$	HGR	HOME	HPLLOT
HTAB	IF	INPUT	INT	INTEGER
INVERSE	KILL	LEFT\$	LEN	LET
LIBRARY	LINE	LOADLIBRARY	LOC	LOF
LOG	LONG	LOOP	MID\$	MKDIR
MOUSETEXT	NAME	NEXT	NIL	NORMAL
NOT	ON	ONERR	OPEN	OR
OUTPUT	PEEK	POINTER	POKE	POP
POS	PRAGMA	PRINT	PUT	RANDOM
READ	REM	RESTORE	RESUME	RETURN
RIGHT\$	RMDIR	RND	SEEK	SELECT
SETMEM	SGN	SHARED	SIN	SINGLE
SIZEOF	SPC	SPEED=	SQR	STEP
STOP	STR\$	STRING	SUB	TAB
TAN	TCP	TEXT	THEN	TO
TOOL	TOOLERROR	TYPE	UNLOADLIBRARY	UNTIL
USING	VAL	VERSION	VTAB	WAIT
WEND	WHILE			

Don't worry; you don't need to memorize the list. The important thing to remember is that there are some words you can only use in specific ways. If you get strange errors from GSoft BASIC, you can refer back to this table to see if the reason is misusing a reserved word.

Case Sensitivity

BASIC is case insensitive. That means that you can type the reserved words using lowercase characters, uppercase letters, or any mix of case you prefer. For added speed, though, GSoft BASIC always converts everything to uppercase letters. The program is actually converted from the text you type to a shorter internal format called tokens. When you list or edit a program, GSoft BASIC converts these tokens back to text. In the process, it prints everything using uppercase letters and indents to program automatically to indicate the program's structure. Spaces that are not part of a string, like the space in

“Hello, world.”, are removed. As the program is converted from tokens to text, new spaces are inserted between most program symbols.

Where Are The Line Numbers?

In many versions of BASIC, each line must start with a number. Obviously that’s not true in GSoft BASIC, since we didn’t use one, but why the difference?

To understand where the line numbers went you have to understand why they were ever used in the first place. In very old implementations of BASIC there are actually two reasons for using line numbers.

The first use of line numbers has more to do with typing the program than running it. Early versions of BASIC were written for computers that didn’t have much memory. To save space—and programming time!—these implementations of BASIC used simple editors that entered or changed one line at a time, rather than editors like the one in GSoft BASIC that work more or less like a text editor. The old kind of editor is called a line editor. In a line editor you need some way to tell the editor which line you are going to change. Older implementations of BASIC use a number at the start of each line to identify the line. The lines are sorted in numerical order.

The second use of line numbers is to label the line for a statement that jumps to that line, often a GOTO statement. You won’t see statements like GOTO much in this course, so we don’t generally need line numbers for this purpose, either.

Like most modern implementations of BASIC, GSoft BASIC just doesn’t need line numbers on each line, and rarely needs them at all. Since they aren’t needed, you aren’t required to type them. You can still use line numbers, and in fact GSoft BASIC actually has a built-in line editor that works a lot like the old Applesoft BASIC line editor. In this course, though, we’ll assume you’re using the modern full screen editor. We won’t use line numbers unless the program itself needs them.

Problem 1.1. Rewrite the hello world program so it says hello to you. For example, my name is Mike, so I rewrote the program to say "Hello, Mike." Save this program as NAME.

- ◆ Note The disk that comes with this course has all of the programs you see in the text and solutions to all of the problems. Programming is a skill, so you should type the sample programs yourself and try to solve the problems yourself, but if you get stuck, check the solutions disk.

How Programs Execute

With what we know now, we can start to write larger programs. Our first step will be to modify the hello world program to write five lines instead of one. We'll create a program that writes a limerick to the screen.

```
PRINT "There was a young man from Lenore"
PRINT "Whose mouth was as wide as a door."
PRINT "  While trying to grin,"
PRINT "  He slipped and fell in,"
PRINT "And laid inside out on the floor."
```

Type in the program and save it on your program disk as Limerick. Use the RUN command to run the program.

Did the program do what you expected? It does bring up an obvious point. Like sentences in a book, BASIC reads and processes your program in the order it is written. The first line is executed first, the second is executed second, and so on.

Later you'll learn several statements that modify the way a program executes, executing one line or a group of lines several times, for example, but the essential point of this exercise is still critical. Whatever a program is running, statements are executed in a specific, logical order that can be predicted ahead of time.

Problem 1.2. Write a program that prints your name and address. Print the address on separate lines, just as you would on an envelope.

Problem 1.3. With a little work, you can create a readable letter by coloring in squares on a sheet of graph paper. The smallest number of squares that works well for uppercase only letters is seven high by five wide. This is the idea used to form characters on the computer screen from the small dots called pixels.

Write a program to write your first name to the screen in this form. Use the * character to fill in the squares. For example, I would ask the computer to write this to the screen:

```
*   *   ***   *   *   *****
** **   *   *   *   *
* * *   *   * *   *
*   *   *   **   ***
*   *   *   * *   *
*   *   *   * *   *
*   *   ***   *   *   *****
```

Graphics Programs

There's a lot you can do with text, but the Apple IIGS has some stunning graphics, too. It's time to start using some of that power. One word of caution, though: The graphics that are built right into the BASIC language itself are rather limited. While some implementations of BASIC have extensive graphics commands, there is no widely accepted standard set of graphics commands. For the most part, our examples will use the powerful graphics commands of QuickDraw II, the graphics package built into the Apple IIGS itself. As a result, the information in this section that deals with graphics is particular to the Apple IIGS. Other computers may do things a bit differently.

The Apple IIGS has a large number of built-in subroutines to do complicated tasks for you. These subroutines are called tools. They are grouped by function into groups called tool sets. The entire collection is what people refer to as the toolbox. The toolbox is a large and wonderful collection which we won't have time to explore fully, but we will use some of the tools to do some work for us from time to time. Graphics is one of those times. QuickDraw II is one of the tools in the Apple IIGS toolbox.

The following program is our first venture into graphics.

```
HGR
SETPENMODE (0)
SETSOLIDPENPAT (15)

MOVETO (10, 10)
LINETO (45, 10)
LINETO (45, 40)
LINETO (10, 40)
LINETO (10, 10)

INPUT " ";A$
```

Type in this program and save it as Square, then run the program. You will see a square, about one inch high and once inch wide, on your screen. (Depending on the monitor you are using and how it is adjusted, the size of the square may vary a bit.)

One of the things you'll notice right away is that the text you normally see on your screen has vanished. While there are some ways to display both text and graphics at the same time, for the most part you get one or the other. The first line of the program, HGR, switches from the text display to a graphics display which can show 320 colored dots

called pixels on each row. There are 200 rows of pixels on the entire screen. Each of the pixels can be one of 16 distinct colors.

As soon as the program finishes the display switches back to the text screen. The reason you're still looking at the graphics screen is that the program is still running! The last line of the program waits for you to type a string and press return. As soon as you press the return key, the program will finish and the display will switch back to the text display. We'll look at the INPUT statement in more detail later in the course; for now, just use this command whenever you want the program to pause.

All of the remaining lines in the program are actually calls to QuickDraw II, not commands that are a part of BASIC. In each case, the line is the name of one of the commands built into QuickDraw II. This is followed by one or two numbers enclosed in parenthesis. If there are two numbers, these numbers are separated by a comma. This basic format is something you'll see over and over as we explore the Apple IIGS toolbox, GSoft BASIC, and later, program pieces called subroutines that you will write yourself.

The first two drawing commands tell QuickDraw II how you want to draw lines. SETPENMODE (0) tells QuickDraw II to replace any existing dots with new dots. That makes sense, so you might wonder why you need to bother. QuickDraw II can do other things when it draws, so we need to start by telling it to do the simplest of the alternatives. SETSOLIDPENPAT (15) tells QuickDraw II to draw white lines.

The next five lines draw a square in the graphics window. To understand how they work, we need to start by examining the coordinate system used by QuickDraw II. To QuickDraw, the top left dot on the screen is at 0, 0. As you move to the right, the first number increases. In other words, 90, 0 is 90 dots to the right of 0, 0, but on the same line. As you move down, the second number increases. The point 0, 40 is 40 dots below 0, 0. You can use numbers so large they go off of the screen to the bottom or right, or even negative numbers that would theoretically show up above or to the left of the screen. In that case, you can't see the lines, but QuickDraw II will still draw all of the line that is in the window.

The first command to draw the square is MOVETO. It doesn't actually draw anything at all. MOVETO positions the graphics pen over a particular pixel on the screen. The next line, LINETO, draws a line by coloring all of the pixels from 10, 10 to 45, 10. The remaining LINETO commands draw the remaining three sides of the square, coming back around to the original point of 10, 10.

Throughout this section, I've talked about drawing a square, but this program is drawing a shape that is 35 pixels wide and 30 pixels high. Obviously, something strange is happening. The reason for the discrepancy is that pixels on the Apple IIGS graphics screen are slightly taller than they are wide. The exact amount varies a bit, but on my screen, these coordinates produce a square.

There's one other new feature in this program. The program itself carries out three distinct steps: First, it gets ready to draw. Next it draws a square. Finally, it waits for you to press the return key before stopping. It's easy to see these three steps in the program because of the strategic placement of two blank lines to divide the commands into three groups. The blank lines actually take up a small amount of space in the finished program, but the space used is pretty negligible. The extra clarity is well worth the cost of a few bytes of memory.

Problem 1.4. There are a total of sixteen colors that you can use. The `SETSOLIDPENPAT` call is used to choose from these colors. In our example, we used color 15 to draw the square in white. You can use any number from 0 to 15. In fact, you can actually use larger numbers, but that doesn't give you more colors—the same 16 colors are repeated over and over.

Try some of the other colors. Be sure and try color 0. What happens?

Problem 1.5. An equilateral triangle is a triangle where each of the three sides are the same length. Write a new program to draw an equilateral triangle with 1 inch sides. Make the bottom flat, with one point on the top.

Problem 1.6. Modify the program in problem 1.5 to draw a six sided star by drawing two equilateral triangles, one pointed up and one pointed down, and overlapping the triangles. Make the star green.

Problem 1.7. Write your name in the graphics window by drawing lines. If your name has letters with curves, use a few short lines to approximate the shape of the letter.

Apple IIGs Default Graphics Colors

<u>Color Number</u>	<u>Color</u>
0	black
1	dark gray
2	brown
3	purple
4	blue
5	dark green
6	orange
7	red
8	beige
9	yellow
10	green
11	light blue
12	lilac
13	periwinkle blue
14	light gray
15	white

Lesson Two — Variables and Loops

Integer Variables

You have probably heard that computers are very good at dealing with numbers. This is quite true. In this lesson, we will start to use numbers and variables in our programs. If you aren't a math whiz, though, don't panic. We won't be dealing with anything more complicated than simple arithmetic in this chapter.

Let's start by typing in this program.

```
REM This program prints a table of numbers and squares of the
numbers

I% = 1
S% = I% * I%
PRINT I%, S%
I% = I% + 1
S% = I% * I%
PRINT I%, S%
I% = I% + 1
S% = I% * I%
PRINT I%, S%
I% = I% + 1
S% = I% * I%
PRINT I%, S%
I% = I% + 1
S% = I% * I%
PRINT I%, S%
```

- ◆ **Note** Sometimes a line in a BASIC program is too long to fit on one line in this book. When that happens, the second and subsequent lines are further to the left than the rest of the program. When you type the program, put everything on one line in the editor. In this sample, the line “numbers” is actually a continuation of the first line; “numbers” should appear at the end of that line.

One of the first things you will see in our program is a comment. Comments are a special kind of command that doesn't do anything. The comment starts with the

command name, REM. Everything after this command name, all the way to the end of the line, is ignored. You can always leave a comment out entirely, and the program will do exactly the same thing as it did when the comment was there. Why, then, do we bother?

If your memory was as good as the computer's, and if no one else ever read your programs, you wouldn't need comments. Comments are for your benefit, as well as the benefit of all those poor lost souls who will have to figure out what you did later. One good place to put a comment is at the beginning of the program, identifying quickly what the program is for. It's not a bad idea to put your name and the date the program was written there, too. As you get used to seeing comments in the examples, you'll find that comments also help at the start of each logical section of the program—each section of lines that do one conceptual thing.

There are actually two comment commands in GSoft BASIC. The REM command you saw in the example program is pretty much universal in BASIC, but it takes three characters, and some people think it looks a bit ugly. GSoft BASIC lets you use an exclamation point instead of the characters REM. While using an exclamation point to start a comment is hardly universal in BASIC, it's not uncommon in other implementations, either.

Using an exclamation point, the comment looks like this:

```
! This program prints a table of numbers and squares of the numbers.
```

It works exactly the same way as the first example.

Computers can work with a vast array of number formats, each of which has a special purpose. The two most common number formats are integers and reals. Integers are whole numbers, like 4, -100, or 1998. Real numbers include the numbers between the whole numbers, like 1.25 or 3.14159.

The memory of a computer is made up of a vast series of numbers, but in a language like BASIC, we don't have to deal with them the same primitive way the computer does. Instead, we can define variables. A variable is just a place where you can put a value. We use two variables in our program; they are called I% and S%. Within certain limits we can put any number we like in these variables. It's exactly like putting two names for numbers on a sheet of paper and continuously erasing the number to replace it with a new one.

The first thing we need to do is learn to put a number in a variable. We do this with something called an assignment statement, which is sometimes called a let statement. The line

```
I% = 1
```

tells the computer to place the number 1 in the variable I%. The = character is called the assignment operator. The very next line puts this value to use.

```
S% = I% * I%
```

Here, we multiply I% by itself and put the result in the second variable, S%. The * character is used in computer languages for multiplication because a computer would confuse x in "I% x I%" with a variable named X. The result is saved in the location named S%. Finally, we write the values.

```
PRINT I%, S%
```

The PRINT statement deserves a little more attention, since there are several new concepts here. We have already used the PRINT statement to write characters to the screen. In this case, though, we are writing two numbers. Any time we use the PRINT statement to write two things, the two things are separated by a comma or a semicolon. If you separate the values with comma characters, BASIC separates the values into neat, tabbed columns. Semicolons are used when you don't want extra spaces or columns, as in

```
PRINT "That will cost $"; MONEY
```

As you can see, we can also mix strings and numbers in the same PRINT statement.

The rest of the program should make sense at this point. BASIC reads the program one line at a time, in the same order you do, and does what the line tells it to do right after the line is read. It does this until it reaches the end of the program, then stops.

More About Variable Names

If you recall, I said BASIC could use several kinds of numbers, like integers and real numbers. So which are these? Both I% and S% are integers, so they are limited to whole numbers. There are two ways to tell BASIC what kind of number to use. The first is to follow the name of the variable with a special character. For integer values, that character is %, so I% is an integer variable. For real variables, you can use a ! character, but BASIC also creates a real variable if you don't use any character at all at the end of the variable name.

Interestingly enough, that last character counts. It's perfectly legal to have variables named I%, I and I! In the same program, and each of these holds a distinct value. The

first variable is an integer, while the last two are real numbers. On the other hand, not everything that is possible is a good idea. In most cases it's best to use distinct names. It makes the program easier to understand, and as your programs get longer, that will become very important.

As for the names of the variables themselves, they pretty much follow the same rules you would use for writing words, so long as you don't pick one of the reserved words listed in Lesson 1. Each variable name starts with an alphabetic character or the underscore character, `_`. The rest of the name can be any number of alphabetic characters, underscore characters and digits, while the last character can also be one of the type characters, like `%` for integers. The case of the characters does not matter—you can use the name `S` in one place, and `s` in another, for example. BASIC treats the names as the same variable, and in fact, it will change all of the lowercase letters to uppercase before printing them.

What all of those technical rules really amount to is that you can use anything that looks like a word as the name of a variable. You can also use numbers as long as a digit isn't the first character, perhaps naming a series of related variables `COST1`, `COST2`, and so forth. The underscore character is usually used when you want to stuff two English words together to form a variable name. You can't use a space character, so you use the underscore instead, as in `CURRENT_INTEREST_RATE`.

Problem 2.1: The Fibonacci series is a sequence of numbers obtained by adding the two previous numbers in the series. The series starts with 0 and 1. Write a program with three variables named `LAST%`, `CURRENT%`, and `FIB%`. Set `LAST%` to 0 and `CURRENT%` to 1.

Now do the following steps five times:

1. Compute `FIB%` by adding `CURRENT%` to `LAST%` and saving the result in `FIB%`.
2. Print `FIB%`.
3. Assign `CURRENT%` to `LAST%`.
4. Assign `FIB%` to `CURRENT%`.

The result should be the numbers 1, 2, 3, 5 and 8, all on a different line.

Fibonacci numbers seem to occur frequently in nature; no one is quite sure why. The number of petals in a flower and the number of leaflets on a compound leaf are often Fibonacci numbers.

Using DIM To Declare a Variable Type

If you remember, I said there were two ways to tell BASIC what type of number you want to store in a variable. The first is to follow the variable name with a special character, like % for integer values. The second way is to use a DIM statement. The DIM statement is generally used to create arrays in BASIC, and we'll use it for that later in the course. For now, though, we'll put it to the more mundane use of making a variable hold the kind of number we want without the need for special characters.

Here's how we can create a variable named I, but make it an integer variable instead of a single-precision floating-point variable, like it would be if we did not use the DIM statement.

```
DIM I AS INTEGER
```

You can also use SINGLE instead of INTEGER to declare a single-precision real variable, or STRING for a variable that holds a string of characters. Later on we'll start using other kinds of numbers. All have a named type that you can use in the DIM statement to create variables.

There are two schools of thought on whether to use the special characters or DIM statements to declare variables. The special characters make it pretty obvious what kind of value goes in the variable, and it also lets you start using variables without the hassle of creating a DIM statement to declare the variable first. That's one of the many things that makes it easier to write a short program in BASIC than in, say, C or Pascal. On the other hand, declaring all of the variables at the top of the program is a nice way to start a large program or subroutine. It gives you a chance to document what the variable is for with a comment, like this one:

```
DIM S AS INTEGER :! Square of the number
```

So which is better? Well, it depends. Personally, I use type characters for short programs and for programs that I write once to do a specialized task, then throw away. I use DIM statements and careful comments on longer programs. I'd suggest trying both methods to see which you like. You'll see both in this course.

If you look closely, I slipped in one other new idea in this example. The : character is used to put two statements on the same line. We need it here because the DIM statement and the comment statement can't fit on the same line without it. You can actually use the : character to separate almost any two BASIC statements, but in most cases that makes the program harder to read. This is about the only situation where you'll see the : statement separator used in this course.

The FOR Loop

So far all of our programs have executed one statement at a time, starting with the first and proceeding to the last. In our last sample and problem this started to get a little tedious, as we repeated the same thing over and over, incrementing a number by one each time. Computers are real good at doing tedious things, but most people are not. The FOR loop is the first in a series of statements we will look at that help remove some of the tediousness of writing a program.

Type in the sample program below and run it. Take a crack at figuring out what it is doing on your own before you read further.

```
REM Draw a fan shape in the graphics window

DIM I AS INTEGER :! loop/index variable

! Set up the graphics screen
HGR
SETPENMODE (0)
SETSOLIDPENPAT (2)

! Draw the fan
FOR I = 1 TO 25
  MOVETO (160, 70)
  LINETO (I * 12 - 10, 10)
NEXT

! Wait for a return
INPUT "";A$
```

Most of the things in this program should be familiar by now, although some of them are being used in new ways. The only thing that is really new is the FOR statement itself. In BASIC, we use a FOR loop whenever we need to do something a specific number of times. This could be calculating ten values, or drawing twenty-five vanes of a fan, as our program does.

The FOR loop starts with the reserved word FOR, followed by an assignment. In our case, we are starting our FOR loop with I set to 1. The two statements right after the FOR loop get executed once with I set to 1. What happens is exactly the same as if we substitute 1 for I in the statements, like this:

```
MOVETO (160, 70)
LINETO (1 * 12 - 10, 10)
```

It doesn't stop there, though. When NEXT is executed, the loop starts over with the next value. I is set to two, and the statements are executed again. This continues until I is twenty-five. After executing the statements one last time with I set to twenty-five, the program moves on to the line after the FOR loop.

Problem 2.2: Our first sample in this chapter created a table of numbers and squares. It did this in a fairly clumsy way, by using separate statements to step from 1 to 5. Rewrite this sample using a FOR loop.

Problem 2.3: In the last chapter, we drew a square by drawing its sides with constant integers. We could also draw a rectangle using variables, like this:

```
TOP = 10
BOTTOM = 70
LEFT = 10
RIGHT = 100
MOVETO (LEFT, TOP)
LINETO (RIGHT, TOP)
LINETO (RIGHT, BOTTOM)
LINETO (LEFT, BOTTOM)
LINETO (LEFT, TOP)
```

Use a FOR loop to draw five rectangles, one inside the other. Set TOP, BOTTOM, LEFT and RIGHT before the FOR loop starts. Inside the FOR loop, draw the rectangle, then add six to top and left, and subtract six from bottom and right.

Use DIM statements with appropriate comments to declare TOP, BOTTOM, LEFT and RIGHT as integers.

Some Thoughts on Comments

You may notice more and more comments slipping into our programs. As the programs get longer and more complicated, you will see the trend continue.

The primary use of comments is to describe in plain English what the program is doing. Looking back at the Fan program, the FOR loop is labeled with a comment that says the loop draws the fan. These are a great help. You can read the statements by now, and you know what each one does. No one has to tell you what MOVETO(50, 70) does, for example. On the other hand, it is certainly not obvious to me that these lines of code draw a fan shape. The comment tells me that, and suddenly the purpose behind the

statements is clear. You've also started to see comments used to describe how a variable is used in a program.

The way you comment differs from one language to the next. In assembly language and some high-level languages I like to put comments like these at the right side of the page, lined up in a column. This lets me read the comments quickly, without reading the program. This doesn't generally work well in BASIC because the language reformats your programs for you, taking out extra spaces you insert and putting in spaces where it wants them. This messes up comments that are formatted in columns to the right of the code. In BASIC and some other high-level languages, I prefer putting comments on a separate line just above the code the comment describes. The extra blank line adds a little emphasis, breaking the program up into logical chunks, more or less like paragraphs are used to break sentences into logical chunks in prose.

There is one tremendous pitfall in commenting, though, and that's when the comments don't match the program. Let's assume that the comments in a program describe something that should work, but the program itself doesn't do exactly what the program describes—perhaps the comment says the code draws a fan, when in reality it draws an array of parallel lines. When you go to debug the program, the natural tendency is to read the comments, not the code. This tendency is so strong that it generally takes less time to debug a program with no comments at all than it does to debug a program with incorrect comments! This is a surprising result, but it's backed up by research.

There are two points to keep in mind as you think about this paradox of commenting. The first is that the comments are a memory jog, and not always an accurate portrayal of what the program does. As Ronald Reagan might have put it if he had been a programmer instead of an actor, "Trust, but verify."

The second important point is that when you change the code, you need to change the comments, too. That seems so painfully obvious that you probably don't think it's a real problem. Trust me, it is, especially when you are rapidly changing a program to fix bugs or make adjustments to improve speed, or to change the way the program looks on the computer screen. Don't get lazy and put off changing the comments until you finish debugging the code—always change the comments as an integral part of changing the program itself.

Operator Precedence

By now you are getting used to the idea that computers step through a program in a fairly orderly way. Statements are executed top to bottom, left to right, the same way you read. Try the following program, but see if you can figure out what will be printed before you run the program.


```
REM A look at operator precedence  
PRINT 1 + 2 * 3
```

There are two perfectly reasonable ways to compute a value from the expression

$$1 + 2 * 3$$

The first is to work left-to-right:

```
1 + 2 * 3  
3 * 3  
9
```

The second is to follow the rules you may remember from algebra class, and do the multiply first.

```
1 + 2 * 3  
1 + 6  
7
```

As you can see from running the program, BASIC uses the same rules as algebra teachers. BASIC was, after all, originally designed by and for physicists, who tend to take a lot of math courses. Not all languages follow these rules; APL, for example, does work left to right. The way a language determines what order to do operations in is called operator precedence. We might as well call it the operator pecking order; it means the same thing. Computer types like to sound official, though, so we better stick to precedence.

The table below shows all of the operators in BASIC. All of the operators on the same line have the same precedence. The ones at the top are done first. If two operators with the same precedence appear together, they are evaluated left-to-right.

You will learn to use most of these operators as the course continues. For now, the important thing is to remember that this table exists. You will need to refer back to it many times.

Operator Precedence in BASIC

@
+ - NOT
^
* /
+ -
= < > <= >=
 <>
AND
OR

If you look closely at the table, you'll see one peculiarity. The + and - operators appear twice. That's because there are technically two different operations they can be used for. The - operator in a statement like

$$X = -X$$

is called a unary operator because it operates on one thing. This is the top - operation. There is also a unary + operation, although it doesn't really do anything. The second form of + and - is the kind you normally think of for addition and subtraction. It's the operation you are using in statements like

$$X = X + 1$$

This version of the operation is called a binary operator.

In our original expression, if you really wanted to compute the value 9, you could have used parentheses. BASIC does all operations inside of parentheses as a group, and uses the result in the rest of the expression.

$$\begin{aligned} &(1 + 2) * 3 \\ &3 * 3 \\ &9 \end{aligned}$$

The Maximum Integer

Growing up with a last name like Westerfield, I quickly learned that computers have limits. It seemed like all of the people who programmed had names like Wirth, or Ritchie, or Steele. All of those silly forms that asked me to put each letter into a separate block

had ten blocks. It upset me: my name isn't Westerfiel, it was Westerfield. The protests of a seven year old are seldom heeded, though.

Computers have become a lot more friendly since then, perhaps in part due to the fellow protests of people like Joe Jabinoslawski. But they still have fixed limits on just about everything. The limit may be very large, but it is there. Integers are no exception. Every implementation of BASIC imposes some upper limit on integers—some largest number that can be stored in an integer variable. On most microcomputer implementation of BASIC, this value is 32767. As with the upper limit, there is a lower limit, too. The lower limit is usually -32768.

These two numbers probably seem like rather odd choices for the upper and lower limit for integer values, but there is a good reason for why these are the limits. It has to do with the way numbers are stored in a computer. We really don't need to delve into that at the moment, though. The important thing is that you know that there is a maximum and minimum.

If you try to stuff a number bigger than 32767 or smaller than -32768 into an integer, different implementations of BASIC handle the problem in different ways. Most, like GSoft BASIC, stop the program with some sort of error message.

While integers that range from -32768 to 32767 are big enough for most uses, there are cases when you need a larger value. Many implementations of BASIC have a special kind of integer that can hold large and smaller numbers. These longer integers are called long integers. You can create a long integer by appending & to the name of the variable, or by declaring the variable as a LONG in a DIM statement.

```
REM Try out a long integer

DIM I AS LONG

I = 500
I = I * I
PRINT I
```

Long integers can hold numbers as small as -2147483648 and as large as 2147483647.

Real Numbers

As everyone knows, programmers drive Porches. At least, many of the folks I meet seem to have that impression. I have never met a programmer that drove a Porche myself. Still, you may be aspiring to high goals, so let's see how long you will be paying off your dream car. We will assume that you want a new car, but not necessarily a fancy one.

We'll spend \$40,000 on our car. We'll assume that you know a banker real well, and can get your car loan at 7% APR, which works out to a monthly interest rate of about 0.58%. That would make the initial interest payment for the first month

```
40000 * 0.0058
$232.00
```

Let's assume you are generous and want to pay \$500 a month. The program below finds out how many months you will be paying.

```
REM Why I don't own a Porche

COST = 40000.0:! initial cost of car
APR = 7.0:! annual percentage rate
PAYMENT = 500.0:! monthly payment

DIM MONTH AS INTEGER :! number of months that have gone by
DIM PRINCIPAL AS SINGLE :! amount left to pay

! no payments made, yet
MONTH = 0

! we start owing this much
PRINCIPAL = COST

! keep going until we're out of debt
WHILE PRINCIPAL > 0.0

    ! count the months as they go by
    MONTH = MONTH + 1

    ! add interest to what we owe
    PRINCIPAL = PRINCIPAL + PRINCIPAL * APR / 100.0 / 12.0

    ! make the payment
    PRINCIPAL = PRINCIPAL - PAYMENT

    ! print how we're doing
    PRINT MONTH, PRINCIPAL
WEND
```

The negative number after the last payment shows that you didn't quite have to pay \$500.00 the last month to pay off the loan. The number of months this takes shows why I own a Toyota. An old one.

This program builds on your previous knowledge, but it also introduces a wealth of new ideas.

The first is a completely new way to loop over a group of statements. The WHILE loop executes all of the statements between the WHILE and the WEND that ends the loop for as long as some condition is true. In our while loop,

```
WHILE PRINCIPAL > 0.0
```

the condition is that PRINCIPAL must be greater than zero. The > character is a comparison operator. It compares the number to the left of the operator to the number to the right of the operator. If the left-hand number is bigger than the right-hand number, the result is true. If the left-hand number is smaller than or equal to the right-hand number, the result is false. The loop continues to execute the statements as long as the condition is true. In our program, the program continues until the car is paid off, at which time the principal is less than zero or equal to zero.

There are a total of six comparison operators. The table below lists the operators and what they test for.

operator	test for...
A < B	A less than B
A > B	A greater than B
A <= B	A less than or equal to B
A >= B	A greater than or equal to B
A = B	A equal to B
A <> B	A not equal to B

FOR loops and while loops have much in common. Both are used to execute a group of statements more than one time. In the case of the FOR loop, though, we must know how many times the loop will be executed before we start. In the case of the while loop, we can loop until some condition is satisfied, without knowing in advance how many times through the loop it will take to satisfy the condition.

PRINT USING for Dollar Amounts

One of the problems with real numbers is that they may be a little more exact than we want. In the Porche example, some of the dollar amounts show fractions of cents. What we'd really like to see in most situations is the amount rounded to the nearest cent.

BASIC uses a variation on the PRINT statement to handle situations where we want more control over the way numbers are printed. It's called the PRINT USING statement because you print the amounts using a format string. The PRINT USING statement prints the format string, but as it does, it looks for special sequences of characters called format models. A format model tells BASIC how to print a value. A value that comes after the format string is printed for each format model.

In our case, we might want to print something like

```
After 100 months, the amount owed is $3933.06.
```

You accomplish this with the statement

```
PRINT USING "After # months, the amount owed is $####.##.";  
MONTH, PRINCIPAL
```

There are an enormous number of ways to create format models, but all of the format models used for numbers are based on a series of # characters. Each # character reserves one character in the final output for the number. A decimal point appearing in the format specifier shows how to handle fractional digits. In our example, the two # characters appearing to the right of the decimal place tell BASIC to print exactly two digits to the right of the decimal point. The value is rounded to two digits if there are more than two digits available.

Integers and real numbers behave a little differently if there isn't enough room to print the entire value. Our program shows how integer values work quite well. We only left one space for the number of months elapsed. When the value hits 10, and later 100, the program prints the entire value anyway.

Real numbers are handled a bit differently. Instead of adding characters to handle a larger number, the # characters themselves are printed. This seems mighty strange. Why not just expand the number of characters for real numbers, too? Actually, the reason has to do with the possible size for real numbers as compared to integer numbers. Even a long integer is, at most, 10 characters, so printing the entire number doesn't cause any real problems. Real numbers can be considerably bigger, though. One kind of real number that you'll get acquainted with later could use over 300 characters to print a

number! Expanding a field automatically to handle 300 characters when you expected just a few can be very annoying.

If you allow more room than is needed the extra space is still used. Blank characters are inserted to fill in the space. You can see how this works if you change the Porche sample to use this new PRINT USING statement. If you don't want to type the entire program, run Porche2 from the samples disk.

There are several special characters you can use in a format model, and our example makes use of one of them. BASIC programs need to print dollar amounts on a regular basis, so the \$ character gets special handling. When you replace the first two # characters with \$ characters the PRINT USING statement prints the \$ character immediately to the left of the first number. Any extra spaces needed to fill out the format model appear to the left of the \$ character.

Like I said, there are an enormous number of variations on format models. We'll cover a few more as they come up in this course. For a complete run-down, see the reference manual that comes with GSoft BASIC.

Problem 2.4: Modify the sample program to find out how big the payments need to be to pay off the car in four years.

Hint: Start with a payment of \$900, then increase or decrease the payment to get to a solution. You are playing a guess-the-payment game. If you pay off the loan in less than 48 months, or if you need to pay a lot less than the payment on the 48th month, you need to decrease the payment size. If it takes longer than 48 months, make the payment larger. You should only go to the nearest cent. The amount will not work out exactly.

Problem 2.5: Let's assume that you are working with the planning board of the local city government. You live in a pleasant city, but due to the local geography, the city can't expand indefinitely. You don't want the city to become too crowded, either. The current population size is 30,000 people. Everyone seems to agree that if the city gets any bigger than 50,000 people, it will be overcrowded.

One councilman has proposed new legislation to prevent the city from growing at more than 10% per year. At this rate, how long will it be before the city hits the limit of 50,000 people? Use a program very much like the sample program, but with a growing population instead of a shrinking principal to find out. Do you feel this is acceptable?

This is not an idle problem. While the numbers were different, this is exactly the situation faced several years ago by the city of Boulder, Colorado. The answer they found caused some changes in the thinking of the city planners, and affected the outcome of some zoning legislation.

Problem 2.6: Inflation has been running at about 4% for the past few years. On average, then, something that costs \$1.00 at the beginning of the year will cost \$1.04 by the end of the year. Assuming a gallon of gas costs \$1.00 today, what will it cost in ten years if inflation continues at 4%?

A few years ago inflation was running at about 12%. Try this inflation figure. Is this rate a problem?

Exponents

Integers were limited to a specific size. Real numbers have limits, too, but the limits are of a slightly different nature. This is because real numbers use exponents to represent very large and very small numbers.

Exponents are the computers way of dealing with something called scientific notation. An exponent is a power of ten that follows the real number. For example,

$2.5E2$

means 2.5 times 10 raised to the power of 2. You can also think of the power as the number of zeros to add to the 1. Ten to the power two is 100, for example. One-hundred times 2.5 is 250, so $2.5E2$ is 250.

Exponents can also be zero. An exponent of zero means a 1 with no zeros, or just 1. Multiplying by one gives the original number, so $2.5E0$ is just 2.5.

Finally, exponents can be negative. A negative exponent means to divide by ten to the indicated power, so $2.5E-3$ means to divide 2.5 by 1000, giving 0.0025.

A quick way to work with exponents is to move the decimal point to the right for positive exponents, or to the left for negative exponents.

Real numbers can get quite large and quite small, but there is a limit to the size. In GSoft BASIC real numbers can have exponents in the range $1E-38$ to $1E38$. There is also a limit to the number of digits that can be handled. It's a lot like a calculator with a ten-digit display. If you need numbers with more than ten digits of accuracy, you have to get a different calculator. GSoft BASIC real numbers have seven digits of accuracy.

Like many implementations of BASIC, GSoft BASIC also supports another type called DOUBLE. Double values are handled just like real values, but they can have bigger exponents and are more accurate. In GSoft BASIC double values can have exponents that range from $1E-308$ to $1E308$, and can display seventeen digits accurately.

The following example shows how to use real numbers to represent very large numbers.


```
REM There are about 6 billion people in the world. Assuming
REM a growth rate of 1.8% per year, how many people will there
REM be in 100 years?

DIM PEOPLE AS SINGLE :! number of people
DIM YEAR AS INTEGER :! current year

PEOPLE = 6E9
FOR YEAR = 1 TO 10
    PEOPLE = PEOPLE * 1.018
NEXT
PRINT "At 1.8% growth, there will be ";PEOPLE;" people in 10
years."
```

These numbers are about right for 1998. Pretty scary, isn't it?

Problem 2.7: Some germs can reproduce every twenty minutes. They reproduce by fission, where one germ splits in half to make two new germs. Assuming nothing stopped their growth, how many germs would there be after one day, starting with a single germ?

Why So Many Kinds of Numbers?

So far you've seen three kinds of numbers, and double-precision floating-point numbers have been mentioned but not used. There's even a fifth kind, called a byte.

It's fair to ask why there are so many kinds, and, more important, when each kind should be used.

There are three competing issues that force us to use so many kinds of numbers. The first is space. An integer uses two bytes of storage; a long integer and single-precision floating-point number both use four bytes of storage, and a double-precision floating-point number use eight bytes of storage. A byte is a unit of storage that, on most computers, is made up of eight on or off switches whose values are represented by a 0 or 1; each of these is called a bit. The important point for us BASIC programmers, though, is that we can store two integers in the same space it takes to store one long integer or single-precision floating-point value, and of course a double-precision floating-point value takes up four times the space of one integer. Size becomes important when your programs use large databases that are made up of thousands or millions of numbers. Size is also important when you're waiting for a program to scan large disk files, or when you're trying to cram a large file onto a disk or send it over a network.

The second issue is speed. Multiplying two long integer values generally takes about four times as long as multiplying two integer values; the same is true when you compare

double-precision values to single-precision values. And a math operation on single-precision floating-point values takes longer than the same operation on long integers, even though they are the same size. In many programs, speed isn't that big of an issue—but in some it is, and when speed is important, it's important to use the fastest operations available. Like size, speed concerns dictate that we use integers where possible, selecting long integers next. From a size standpoint, long integers and single-precision floating-point values come up as a draw, but from a speed standpoint we choose long integers. And of course, double-precision floating-point values are the slowest and take the most room, so we want to avoid them whenever possible.

But it isn't always practical to use integers. After all, when you're calculating population growth, car payments, or the area of a circle, you need to use values that might not be an integer. Many scientific and statistical calculations simply need more precision—or a greater range of exponent—than single-precision floating-point values can deliver. There are even a few odd-ball algorithms in the field of numerical analysis that work best when you use two different sizes of floating-point numbers at various points in the calculations.

So, in a nutshell, the reason there are so many kinds of numbers is that you, as the programmer, are balancing contradictory goals. You need to write programs that are fast and use as little space as possible, but you also need to use numbers that give you an accuracy and range of values that will handle the situation.

To sum up the rules, pick numbers this way:

1. If you can, use integer variables. They are the smallest, and calculations with integer values are always the fastest.
2. If you will be dealing with values that are too large or too small for an integer variable, switch to long integers.
3. If you need values that are not whole numbers, or if the values are too large or too small for a long integer, use single-precision floating-point numbers.
4. If you need more digits of precision or a larger range of exponents than single-precision floating-point can deliver, switch to double-precision floating-point.

There are specialized tools for dealing with situations where even double-precision floating-point numbers can't cut it. We won't deal with them in this course, but if you want to branch out, look for articles dealing with so-called infinite precision math packages.

Lesson Three – Input, Loops and Conditions

Input

So far, all of your programs have only done one thing. No matter how many times you ran the program, unless you changed the program itself, it always did the same thing. The reason, of course, is that the programs could never ask you for any information. It's time to start controlling our programs a bit more through the use of input.

Your first program was a pretty simple one; it used the PRINT statement to write a string to the screen. You have already learned to write integers and real numbers using PRINT. BASIC uses the INPUT statement in much the same way to read numbers and strings.

Actually, you've already used the INPUT statement in a limited way. When a graphics program finishes, the display switches back to the text screen, which doesn't give you a chance to look at the completed drawing. We've been using the statement

```
INPUT " "; A$
```

to force the program to wait for you to press the return key, giving you a chance to examine the graphics screen before the program finishes.

You can experiment with the INPUT statement to quickly get an idea how it works. From the GSoft BASIC command line, type

```
INPUT A
```

and press the return key. A question mark shows up on the screen, telling you that BASIC is waiting for some kind of input. Type a number and press return, then try the command

```
PRINT A
```

As you can see, the number you typed is entered into the program.

For a short, quick program, this is perfect. Here's a simple command that places a prompt on the screen so you know it's ready for you to type, lets you enter a number, and stores the result in a variable. It works with integers, long integers, and both single-precision and double-precision floating-point numbers. It even works with strings,

something you probably guessed from the INPUT statement we used to pause at the end of graphics programs.

There are a lot of situations where this simple approach is appropriate, but as your programs get longer, this simplicity doesn't always work. The first situation that pops up is using something other than a question mark as a prompt. If you wrote the program, will use it a few times, and throw it away, the question mark is fine, but if other people will use your program or if you'll use it later, it's a good idea to enter something more informative. INPUT allows you to change the prompt by following the command name with a string and a semicolon. The string is used as the prompt. If you use an empty string, like we did in the graphics programs, no prompt is printed at all.

Try this program to see how prompts work.

```
INPUT "Please type your name: "; NAME$  
PRINT "Hello, "; NAME$
```

There is one last feature of the INPUT statement that is pretty handy for short programs, but tends to get in the way in longer ones. You can read several values with a single INPUT statement, and the person using the program can reply with more than one value on the same line. In each case, the variables or values are separated by commas.

Here's a program that reads two pairs of numbers, draws a line between the points, and waits for a final press of the return key before quitting. It puts all of these ideas to use.

```
INPUT "First coordinate (enter x, y): ";X1, Y1  
INPUT "Second coordinate (enter x, y): ";X2, Y2  
HGR  
MOVETO (X1, Y1)  
LINETO (X2, Y2)  
INPUT " ";A$
```

One of the peculiar things about the INPUT statement is that you don't have to enter the values exactly when the program expects them. Normally, you'd expect to see something like this as you type your responses:

```
First coordinate (enter x, y): 1,1  
Second coordinate (enter x, y): 100,100
```

That certainly works. But now try entering all of the numbers on the same line, like this:

```
First coordinate (enter x, y): 1,1, 100,100
Second coordinate (enter x, y):
```

Strangely enough, that works, too. If you add a fifth value, the last INPUT statement picks it up, so your program doesn't pause at all!

You can also do just the opposite. If you enter a number, then press the return key, the program accepts the first value, then waits for another—but without showing a prompt.

Our First Game... er, Computer Aided Simulation

Well, let's have some fun. Now that we can hold a simple conversation with the computer we can write our first simple computer game.

```
REM Guess a number
REM
REM This game randomly selects a number from 1 to 100, then
REM lets a player guess the number.

DIM VALUE AS INTEGER :! The value the player will guess.
DIM I AS INTEGER :! The player's guess.

! Introduce the game
PRINT "In this game, you will try to guess a number from 1"
PRINT "to 100. I will tell you if your guess is too high"
PRINT "or too low."
PRINT

! Pick a number from 1 to 100.
VALUE = 1 + RND (1) * 100

! Guard against overflows to 101.
IF VALUE = 101 THEN
    VALUE = 100
END IF

! Let the player guess the number.
DO

    ! Get the player's guess.
    INPUT "Your guess: ";I

    ! If the number is too high, say so.
```

```

IF I > VALUE THEN
    PRINT I;" is to high."
END IF

! If the number is too low, say so.
IF I < VALUE THEN
    PRINT I;" is to low."
END IF
LOOP WHILE I <> VALUE

! If we get here, the number was correct.
PRINT I;" is correct!"

```

There are a lot of new concepts in this program, and we will spend a lot of time examining it in detail, but first type it in and run it.

The DO-LOOP

One of the new things in our program is a pair of new statements called the DO-LOOP statements. This is the third looping statement you have learned in BASIC. The first two, of course, are the FOR loop and the WHILE loop. The DO-LOOP statements are also the last looping statement in BASIC! You're getting there...

Like the WHILE loop, the DO-LOOP statements loop until some condition is satisfied. Unlike the WHILE loop, the condition appears at the end of the loop. (There are some exceptions; we'll discuss those a bit later.) This means that the statements in the DO-LOOP statements are always executed at least one time, while the statements in the WHILE loop can be skipped altogether. This is an important difference, and the key to why there are two loops instead of just one. To understand this difference, let's look at WHILE loops and DO-LOOP statements from some of our programs and compare the two.

In the last lesson, we wrote a program that showed how many payments were needed to buy a car. It contains this loop:

```

! keep going until we're out of debt
WHILE PRINCIPAL > 0.0

    ! count the months as they go by
    MONTH = MONTH + 1

    ! add interest to what we owe
    PRINCIPAL = PRINCIPAL + PRINCIPAL * APR / 100.0 / 12.0

```

```
! make the payment
PRINCIPAL = PRINCIPAL - PAYMENT

! print how we're doing
PRINT MONTH, PRINCIPAL
WEND
```

In this case, we needed to loop until the amount we needed to pay off was zero. It would be possible, although in this case not very likely, for the principal to be zero before the loop was ever executed. This is the key test for a WHILE loop: you must ask yourself if it is possible for the condition that stops the loop to be true before you start. In other words, you want to know if it is possible that you may not want to execute the statements in the loop at all. If that is the case, a WHILE loop should be used.

The DO-LOOP statements look very similar. The only real difference is that the condition is evaluated at the end of the loop, not the beginning.

```
! Let the player guess the number.
DO

! Get the player's guess.
INPUT "Your guess: ";I

! If the number is too high, say so.
IF I > VALUE THEN
  PRINT I;" is to high."
END IF

! If the number is too low, say so.
IF I < VALUE THEN
  PRINT I;" is to low."
END IF
LOOP WHILE I <> VALUE
```

The DO-LOOP statements are generally used in cases where the condition doesn't make sense until after the statements in the body of the loop have been executed at least one time. For example, it would seem to make sense to use a WHILE loop that looks like this to do the same job:

```
! Let the player guess the number.
WHILE I <> VALUE
```

```

! Get the player's guess.
INPUT "Your guess: ";I

! If the number is too high, say so.
IF I > VALUE THEN
    PRINT I;" is to high."
END IF

! If the number is too low, say so.
IF I < VALUE THEN
    PRINT I;" is to low."
END IF
WEND

```

There is a flaw in this code, though. The value of I has not been set when the condition is tested the first time. In this particular case, you might feel safe. After all, you might know that the value of a BASIC variable is always initialized to zero, and zero isn't one of the possible values for VALUE. Depending on this sort of information is a really bad idea, though. First, you may end up moving this program to another implementation of BASIC someday, and that implementation may not initialize values to zero. Most versions of BASIC follow this rule, but there is no BASIC standard that forces everyone to initialize the value of variables. More important, you may pluck this loop out of the original program and insert it into a new one, or add new features to the existing program so that I does have a value other than zero when the loop starts. This sort of change happens more often that you'd think. And when it does, you're left scratching your head, wondering why a part of the program that used to work suddenly starts to fail.

There is a way to rescue the WHILE loop, though. You can start off by initializing I to a number different from value, like this:

```

! Let the player guess the number.
I = VALUE - 1
WHILE I <> VALUE

! Get the player's guess.
INPUT "Your guess: ";I

! If the number is too high, say so.
IF I > VALUE THEN
    PRINT I;" is to high."
END IF

! If the number is too low, say so.

```



```
IF I < VALUE THEN
  PRINT I;" is to low."
END IF
WEND
```

This will work; the test will always fail the first time, so the person guessing the number always gets at least one chance to guess the number. It's perfectly safe, too: It won't fail if you change the program later and set I to some value, or if you run the program on a version of BASIC that doesn't initialize variables to 0. On the other hand, the DO-LOOP statements work, too, but they don't require that you set the initial value before you start into the loop.

The acid test for when to use the DO-LOOP statements, then, is whether or not the test that ends the loop makes sense before the statements in the loop have been executed one time. In our example program, the test uses the value of I, which is read in inside the loop. The test doesn't make sense until the number has been read at least one time, so we use the DO-LOOP statements.

The Flexible DO-LOOP Statement

The DO-LOOP statements are actually more flexible than I've let on. You can actually put the condition at the top or bottom of the loop. As an example, here's our WHILE loop that calculated car payments, reworked to use DO-LOOP statements.

```
! keep going until we're out of debt
DO WHILE PRINCIPAL > 0.0

  ! count the months as they go by
  MONTH = MONTH + 1

  ! add interest to what we owe
  PRINCIPAL = PRINCIPAL + PRINCIPAL * APR / 100.0 / 12.0

  ! make the payment
  PRINCIPAL = PRINCIPAL - PAYMENT

  ! print how we're doing
  PRINT MONTH, PRINCIPAL
LOOP
```

You can also put a condition at both the top and bottom of the loop, or, for that matter, not use any condition at all. If there is no condition at all the loop continues until something else forces the program to stop.

If the condition appears at the top of the loop the DO-LOOP statement doesn't really offer anything that the WHILE loop can't handle, so we won't use it that way. Situations where it makes sense to use a condition at both the top and bottom of the loop, or no condition at all, just don't come up that often. In this course, we'll only use the DO-LOOP statements with a test at the end of the loop.

Of course, it's fair to ask the opposite question. If the DO-LOOP statements can do everything the WHILE-WEND statements can do, why use WHILE-WEND? For that matter, why is it even in BASIC?

I'll speculate a bit here. Neither DO-LOOP statements nor WHILE loops were in the original version of BASIC. They weren't common in BASIC until the structured programming craze hit in the mid 1980's. While I don't know this for a fact, it appears to me that WHILE-WEND loops were introduced by one set of people, and the DO-LOOP statements by another. Eventually, both statements started appearing in BASIC so all of the old programs would run.

The truth is that you don't need WHILE-WEND loops. I think it makes the program easier to follow if you always use WHILE loops when the condition is tested at the top, and always use DO-LOOP statements when the condition is tested at the bottom, so that's what you'll see me doing in the example programs. That doesn't mean you have to do the same thing. Both ways work; just pick one and stick with it.

Random Numbers

One of the new concepts used in our sample program is the random number. You have probably heard that computers are very precise, and that is certainly true. In our number guessing game, though, the last thing we want is for the computer to be precise. This game just won't be much fun if we know beforehand what number the computer will pick. The program uses something called a random number generator to get around this problem.

A random number generator is basically a way for the computer to generate a number, or series of numbers, that seem to be random. Since the computer can only do very specific things, the numbers aren't really random, but they are very hard to predict, and that is good enough for a lot of programs. Since the numbers really aren't random, they are technically called pseudo-random numbers. That's a real mouthful, though, so we will continue to call them random numbers.

We'll write a simpler program to learn more about random numbers.

```
REM A closer look at pseudo-random numbers

FOR I = 1 TO 10
  PRINT RND (1)
NEXT
```

Type this program in and run it. It will print ten pseudo-random numbers. Run it several times, and you'll notice that the numbers are different each time.

One thing that stands out is that all of the numbers are between 0 and 1. Technically, it's possible for the random number generator to return 0, too, but it's very unlikely. It's not possible for RND to return the value 1. That explains why our program can use the lines

```
! Pick a number from 1 to 100.
VALUE = 1 + RND (1) * 100

! Guard against overflows to 101.
IF VALUE = 101 THEN
  VALUE = 100
END IF
```

to pick a value from 1 to 100. If RND returns zero, adding 1 gives a value of 1. If it returns 0.9999999, multiplying by 100 and adding 1 gives 100.99999. Unfortunately, this can cause a number overflow—the number can't be stored exactly, so the computer rounds up, giving a value of 101. That's why the IF statement is used to check for the overflow situation, pushing the value back to 100 if the overflow occurs.

Now make a slight change to the program by adding a new line just before the FOR loop, like this:

```
REM A closer look at pseudo-random numbers

I = RND ( - 1)
FOR I = 1 TO 10
  PRINT RND (1)
NEXT
```

When you run this program, you still get a sequence of ten random numbers. Now run the program a second time and compare the numbers. As you can see, they are the same.

This gives you a solid clue about how random numbers are generated. The fact that you get the same numbers each time you run this program shows that the numbers aren't

really random at all. In fact, each random number is generated by starting with the last number. A complex series of mathematical operations is performed to generate a new number that has no readily apparent relation to the previous number.

In our modified program the first call to RND used a parameter of -1, which told RND to start a new sequence of random numbers using -1 as the starting value. This is called seeding the random number generator; the number is called the seed. All of the random numbers grow from this seed. If you use -2 for this first call you still get a series of random numbers, and they are still the same every time the program runs, but the numbers will be different than the numbers you got using -1 as a seed. In fact, every negative number will perform this same way, generating a consistent series of random numbers, but each negative number generates a series that is different from every other negative number.

We used 1 for the argument to the RND function in the main loop of the program. All positive numbers perform in exactly the same way. They tell RND to generate a random number. It doesn't matter which positive number you use; the mere fact that it is greater than zero tells RND that you want a random number.

There is one last parameter you can use for RND. In a few odd situations, you may want to use the same random number twice. You could do this by saving the random number in a variable and using the value from the variable, but you can accomplish the same thing using 0 for the parameter to RND. When the parameter is 0, RND returns the same value it returned the last time it was called.

It might seem strange to create a predictable series of numbers, but this is very handy when you are testing a program. You can remove the line that seeds the random number generator once the program is finished.

By now, you may realize that the random number generator needs some sort of seed to get the random number sequence started. So how does the first program create a unique series of numbers each time it runs? In GSoft BASIC, if you call RND with a positive argument the first time it is called in a program, the random number generator is automatically seeded from the computer's clock. This frees you from the hassle of finding that first random number to start the sequence. Keep in mind that this service isn't universal in BASIC. You may have to come up with a seed some other way if you use another version of BASIC.

Why Random Numbers Are Important

We will use random numbers in many of our example programs. Random numbers help us to write programs that don't do exactly the same thing each time we use them; that's something we will need over and over again. Here are some places where random numbers are commonly used:

1. Random numbers are used in games like Chess. Games work by scoring moves; the move with the best score is the one the computer makes. If two moves have the same score, random numbers can be used to choose between them so the computer doesn't play exactly the same way each time. In a game like chess or checkers, there are also many good ways to make the first few moves; these are called opening books. Random number generators are used to pick an opening from the opening book.
2. Many dungeon and dragon style computer games work based on probabilities. For example, a character with a particular set of characteristics might have a probability of 0.4 of killing a giant ant with a broadsword. The ant, conversely, might have a 0.2 chance of damaging the player. A random number generator can be used to generate a number between 1 and 100, as our number-guessing game did. The player kills the ant if the number is less than 40. Next, another number is chosen, and the ant hurts the player if the number is less than 21.
3. Computers are often used to do serious simulations. Computer simulations are used to study traffic patterns, wars, and the spread of diseases. As an example, let's assume that you are trying to protect Yellow Stone National Park from forest fires. You could choose to "let it burn," letting nature take its course. You could choose to fight all fires aggressively, but that would lead to a gradual build-up of weeds and wood to burn. You might choose to cut fire lanes through the forest. All of these possibilities can be examined using computer simulations.
4. Random number generators are used in card games to shuffle cards. The random number generator is used to pick which card will be taken from the deck next, taking one card from the remaining cards that have not been dealt.

Problem 3.1. Write a program to throw two six-sided dice twenty times. Use the same ideas used in the number-guessing game. Write the number of spots showing on each of the dice to the screen. Each line should show the value for both dice, like this:

```
1      4
5      2
5      6
```

Write your program so the number of dice and the number of sides are stored in variables, and used throughout the program. This makes the program easy to modify if, for example, you need to roll one 20 sided die instead of two 6 sided dice.

This makes the PRINT statements a little tricky. You can use a print statement like

```
PRINT X, ;
```

to print a value, skip to the next column, and *not* print a carriage return at the end of the line. This allows you to print two numbers on the same line at different places in the program.

Problem 3.2. You can draw a dot in the graphics window by doing first a MOVETO, then a LINETO the same spot. For example,

```
MOVETO (10, 10)  
LINETO (10, 10)
```

draws a dot at 10,10.

Write a program that gradually blackens the rectangle with a left edge of 10, a top of 10, a right edge of 100, and a bottom of 70. Do this using a FOR loop that loops from 1 to MAX, where MAX is set at the top of your program to a value of 5551.

Pick two random numbers inside the FOR loop. The first should be in the range 10 to 100; assign this value to an integer variable called x. The second should be in the range 10 to 70; assign this one to the variable y. Draw a dot at this point using a MOVETO-LINETO sequence.

The result is a program that gradually fills the area with white snow.

There are 5551 dots in the area you are filling, but when the program finishes, not all dots are white. Why?

Problem 3.3. Change the program from problem 3.2 to create multicolored snow by picking the color of the dot randomly. The color should be in the range 0 to 15.

The IF Statement

Computer programs can make decisions. You have already written some programs that use this capability in the form of loops that keep going until some condition is satisfied. In some cases, though, we may only need to do something once, or we may not need to do it at all. That's where the IF statement comes in.

The IF statement evaluates the same kind of condition that you have already used in the WHILE loop and DO-LOOP statements. The condition is followed by the reserved word THEN; this just tells BASIC that you are finished with the condition. Absolutely nothing, even a comment, can appear after the THEN. The IF statement starts a block of statements, just like WHILE-WEND and DO-LOOP. In the case of the IF statement, the block ends with END IF.

If the condition in the IF statement is true, the block of statements between the IF and END-IF statements are executed. If not, the statements are skipped. In a way, the IF statement is like a DO-LOOP statement that doesn't loop.

Let's try a simple example to see how all of this works. In this example, we will use the IF statement to write a program that can count change.

```
REM Count change.

DIM CHANGE AS INTEGER :! The number of cents to count.
DIM AMOUNT AS INTEGER :! The number of a particular coin.

! Get the number of cents to count.
INPUT "How many cents in the change? ";CHANGE

! Write a header.
PRINT "Your change consists of:"

! Count out the dollars.
IF CHANGE >= 100 THEN
    AMOUNT = CHANGE / 100
    PRINT AMOUNT;" dollars"
    CHANGE = CHANGE - AMOUNT * 100
END IF

! Count out the quarters.
IF CHANGE >= 25 THEN
    AMOUNT = CHANGE / 25
    PRINT AMOUNT;" quarters"
    CHANGE = CHANGE - AMOUNT * 25
END IF

! Count out the dimes.
IF CHANGE >= 10 THEN
    AMOUNT = CHANGE / 10
    PRINT AMOUNT;" dimes"
    CHANGE = CHANGE - AMOUNT * 10
END IF
```

```

! Count out the nickels.
IF CHANGE >= 5 THEN
    AMOUNT = CHANGE / 5
    PRINT AMOUNT;" nickels"
    CHANGE = CHANGE - AMOUNT * 5
END IF

! Count out the cents.
IF CHANGE <> 0 THEN
    PRINT CHANGE;" cents"
END IF

```

In this program, each IF statement is used to see if the number of cents left is large enough to give the customer at least one coin of a given size. For example, the first IF statement checks to see how many dollars are in the change. In each block we need to do two things, count the number of coins to give in change and adjust the amount left to give.

The exact order of the calculations is actually quite important. The divide operation returns a single-precision floating-point value. For example, if CHANGE is 70 when the number of quarters is calculated, $CHANGE / 25$ is 2.8. When this value is stored in AMOUNT, which is an INTEGER, the digits to the right of the decimal point are dropped. It's important to realize that the number is not rounded to 3, which is the integer closest to 2.8, but truncated to 2. Storing the single-precision floating-point value in the INTEGER variable AMOUNT has done exactly what we wanted, converting the number to the whole number of coins.

Once we know the number of coins, we can subtract the number of cents we've just counted out by multiplying this whole number of coins by the number of cents in the coin. Following along with the same numbers, the line

```
CHANGE = CHANGE - AMOUNT * 25
```

multiplies the number of quarters, 2 in this case, by 25 to calculate the total amount we just counted out in quarters. This amount is subtracted from CHANGE to give the amount left to count, which is 20 cents.

The ELSE Clause

There are many times when you need to do one thing or another, depending on some condition. In that case, you could use two different IF statements, one after the other, but you can also use an ELSE statement. As a simple example, let's say you are printing the

number of tries it took to guess the number in our number guessing game. It's sort of tacky to print out "1 tries," or worse still, "2 try." With an IF-THEN-ELSE statement, you can print something a bit prettier:

```
IF TRIES = 1 THEN
  PRINT "You guessed the number in 1 try!"
ELSE
  PRINT "It took ";TRIES;" tries to guess the number."
END IF
```

If the condition evaluated by the IF statement is true, the lines between it and the ELSE statement are executed and the lines between the ELSE statement and END IF statement are skipped. If the condition is false, the lines between the IF statement and the ELSE statement are skipped, and the lines between the ELSE statement and END IF statement are executed. This is the model for any either-or kind of situation, where you want to do one thing or another, but not both.

Problem 3.4. Modify the program from Lesson 2 that showed payments for purchasing a car. Allow the user of the program to enter the cost of the car, the interest rate and the number of payments as real numbers. Use an IF statement to see if the payment is larger than the amount of interest that will accumulate in one month. If not, print an appropriate error message. If the payment is large enough, execute the program as it worked before.

The World's Shortest Animation Course

There's one last topic to deal with before we leave the IF statement. We're going to have some fun with it, though, by introducing the topic of computer animation. This section will give you the short version of a course in computer animation. Surprisingly, it covers all of the essential points. Everything beyond what you see in this section is art and technology, not concepts. Admittedly, there's a lot of art and technology out there concerning computer animation, but in the end all of the techniques end up using the same basic principles of moving objects on the screen.

You're almost certainly familiar with the fact that movies, television, and computer animation all work by drawing a series of still pictures at a rapid rate. If the rate is fast enough, your brain interprets the series of still frames as motion. The rate that's used in movies is 24 frames per second; for television and most computer screens it's 30 frames a second.

You can create a very simple animation by simply drawing and erasing a shape in successive positions as it moves across the screen. Here's a sample that moves a square across the graphics screen. It uses a new command, `SETPENSIZE`, to change the size of the dot from the 1 pixel by 1 pixel size you've seen in all of your graphics programs so far to a larger 4 pixel square box.

```
REM Draw a ball sliding across the screen from 0, 0 to 180,
180

DIM X AS INTEGER , Y AS INTEGER :! Coordinates for the ball
DIM I AS INTEGER :! loop counter

! Set up for graphics
HGR
SETPENMODE (0)
SETPENSIZE (4, 4)

! Initialize the ball position.
X = 0
Y = 0

! Animate the ball.
FOR I = 1 TO 180

    ! Erase the old ball.
    SETSOLIDPENPAT (0)
    MOVETO (X, Y)
    LINETO (X, Y)

    ! Compute the new ball position.
    X = X + 1
    Y = Y + 1

    ! Draw the ball at the new position.
    SETSOLIDPENPAT (15)
    MOVETO (X, Y)
    LINETO (X, Y)
NEXT
INPUT " ";A$
```

When you try this program you'll see a lot of problems. The box it erased about as often as it's drawn, which gives a lot of flicker. The box might even seem to vanish for a moment if the timing is just right on your computer. That's because you have two

conflicting actions taking place. At any given time, the video circuitry in your computer is busy drawing some portion of the screen. At the same time your program is busy drawing and erasing the image. If those two activities aren't timed perfectly you can end up erasing the image just before the video circuitry draws that part of the screen, then redrawing it just after it finishes.

There is a simple trick that minimized this problem and, at the same time, lets you draw complex images on a background. It uses a new drawing mode called exclusive OR. Instead of painting a pixel of a particular color on the screen like all of our other programs, this mode actually reverses the color of the pixel. If you are drawing a white square on a black background the effect is identical to what you've already done, but the cool part shows up when you draw the same shape in the same place. Since it's reversing the pixels, not painting them, reversing them a second time erases the object! That, combined with a simple trick of drawing the shape in the new position before erasing it in the old position, improves the image dramatically.

The SETPENMODE command with a parameter of 2 lets you draw in exclusive OR mode; the value of 0 that we've used in all of our programs so far is called copy. Here's a variation on our animation program that puts these ideas to work.

```
REM Draw a ball sliding across the screen from 0, 0 to 180,
180

DIM X AS INTEGER , Y AS INTEGER :! Coordinates for the ball
DIM I AS INTEGER :! loop counters

! Set up for graphics
HGR
SETPENMODE (2)
SETPENSIZE (4, 4)
SETSOLIDPENPAT (15)

! Initialize the ball position.
X = 0
Y = 0

! Draw the ball in the starting position.
MOVETO (X, Y)
LINETO (X, Y)

! Animate the ball.
FOR I = 1 TO 180
```

```

! Draw the ball at the new position.
MOVETO (X + 1, Y + 1)
LINETO (X + 1, Y + 1)

! Erase the old ball.
MOVETO (X, Y)
LINETO (X, Y)

! Update the ball position.
X = X + 1
Y = Y + 1
NEXT
INPUT " ";A$

```

There's still a little flicker. Two things will reduce it further.

First, as your program gets larger and more complicated, it will spend more time calculating various things like relative positions of objects and whether an object has hit another. As long as you do all of this calculation while the object is visible, you increase the chance that the object will be visible when the video hardware draws the portion of the screen it is on.

This still leaves a little flicker, but for many applications it is good enough. The last finesse is more complicated. The very best animation on the Apple IIGS takes the vertical blanking signal into account. This is a notice from the computer that the screen is about to be drawn. As the video hardware draws the screen, the animation software follows along behind, drawing objects in an area where the video hardware isn't busy drawing the screen. While the idea is simple, implementing it is extremely complicated. It is almost always done in very carefully written assembly language programs.

Nesting If Statements

There are some situations where it makes sense to check for more than just one possibility. For example, let's assume that you want to print out a message like "that was your 3rd try." You can print the number of tries, followed by "rd," but that only works for some numbers. You would want to print

```

1st
2nd
3rd
4th
5th

```

and so on. One way to go about it is to print “that was your,” followed by a series of IF statements, followed by printing “try.” The IF statements can be used to decide the suffix for the number of tries. Rather than using a series of separate IF statements, though, you can actually attach another condition right after an ELSE and keep going, as this example shows.

```
PRINT "That was your ";
IF TRY = 1 THEN
  PRINT "1st";
ELSE IF TRY = 2 THEN
  PRINT "2nd";
ELSE IF TRY = 3 THEN
  PRINT "3rd";
ELSE
  PRINT TRY;"th";
END IF
PRINT " try!"
```

The first part works just like all of the other IF statements you’ve seen. If TRY is 1, the condition on the IF statement is true. In that case, the program prints “1st” and skips all of the other possibilities. If TRY is not 1, the next condition is checked. This process continues, checking one condition after another, until one of the ELSE IF conditions is true. As soon as a matching condition is found, the program executes the statement right after that ELSE IF clause and skips the remaining code. If none of the conditions are true, the program executes the lines between the ELSE and END IF statements.

The ELSE clause is optional. If you leave it out, and none of the conditions in the IF or any of the ELSE IF statements are true, all of the other statements are skipped. The ELSE statement should come at the end of the sequence, though, so all of the ELSE IF tests are evaluated.

Problem 3.5. In this problem, you will write a bouncing ball program. You will move a small spot across the graphics screen. When the spot gets to the edge of the screen, it will bounce off.

Start with the animation program from the last section. Before the animation starts, ask the user for a starting x, a starting y, the number of iterations (put this in a variable called ITER), and an x speed and y speed (put these in XSPEED and YSPEED). Check to see if the x and y values are in the graphics window using IF statements. If not, adjust them to be in the window. (The graphics window runs from 0 to 319 horizontally, and 0 to 199 vertically.) Loop over your code to move the ball ITER times.

On each loop, you will need to do the following:

1. Add the XSPEED to X. This moves the ball over.
2. If X is off the screen to the left (less than zero), set it to zero and set XSPEED to -XSPEED.
3. If X is off the screen to the right (greater than 319), set it to 319 and set XSPEED to -XSPEED.
4. Add the YSPEED to Y. This moves the ball up or down.
5. If Y is off the screen to the top (less than zero), set it to zero and set YSPEED to -YSPEED.
6. If Y is off the screen to the bottom (greater than 199), set it to 199 and set YSPEED to -YSPEED.

Be sure to do as much of the calculation as possible while the ball is visible, then quickly draw the ball in the new position and erase the old one.

The practice of writing out the steps for a program in a kind of semi-English form is very useful for designing programs. The roughed-out version of the code is called pseudo-code. It won't run on any computer (at least, none that are available today!), but it helps when you are working on the logic of a program.

A Bit of Iffy History

Like the DO-LOOP statement and the WHILE loop, the sort of IF statement you've learned in this lesson is new to BASIC. It's called a block structured IF statement, and was added to the language at about the same time as the structured loop statements.

While we won't use it in this course, the older form of the IF statement is still a part of BASIC and you'll see it in a lot of books that contain BASIC programs, especially older books. Since you're sure to run across it, we'll take a look at the older form in this section.

The original IF statement was designed as a conditional jump. At that time, every line of a BASIC program had to start with a line number. The line number had to be unique, and the lines were arranged in order by the line numbers. The main way to jump from one place to another was not through the use of structured statements like the loops you've learned, but by using the GOTO statement. The GOTO statement is followed by a line number; control jumps immediately to the specified line. For example, the program

```
10 I = 1
20 GOTO 40
30 I = I + 1
40 PRINT I
```

prints the value 1, not 2, because the GOTO statement jumps past line 30.

The old form of the IF statement works pretty much the same way. You can put a line number right after THEN, and the program will jump to the specified line if the condition is true, skipping to the next line if the condition is false. There is no END IF with this form of the IF statement. Here's a very simple example.

```
10 REM Print all Fibonacci numbers less than 20.  
20 I = 0  
30 J = 1  
40 K = I + J  
50 PRINT K  
60 I = J  
70 J = K  
80 IF I + J < 20 THEN 40
```

There are several variations on this theme. Instead of just THEN, you can use THEN GOTO. You can also replace THEN with GOTO. Regardless of the variation you pick, the statement does the same thing.

The old form of the IF statement is not limited to jumping, though. You can put any statement you like after THEN, and you can even put more than one statement, separated by colons. If the condition is true, all of these statements are executed. If the condition is not true, they are all skipped, and execution picks up with the line right after the IF statement.

Modern implementations of BASIC need to be able to handle both the older forms of the IF statement and the modern block structured form we use in this course. That means they need a way to tell the old form, which is contained on a single line, from the new form, which spans multiple lines. The key is whether anything appears after THEN. If anything at all appears after THEN, GSoft BASIC assumes you are using the old form of the IF statement. If there is nothing at all after THEN, GSoft BASIC assumes the IF statement is the first line of a modern block structured statement. With that in mind, you can see why, earlier in the lesson, there was a warning not to put anything at all after THEN.

Boolean Logic

Most of the IF statements you're likely to use will have a fairly simple condition, like

```
IF TRY = 1 THEN
```

or

```
IF ANGLE < 2 * PI THEN
```

Eventually, though, you'll want to make more complicated comparisons. To do that you'll need three new operations designed for conditional tests and a good understanding of how BASIC actually deals with conditions. These operations are called Boolean operators.

Let's start with a look at how BASIC actually handles conditions. Most of the time you don't really need to know this information, but it is critical if you print a Boolean condition to see whether it is TRUE or FALSE, and it's also occasionally useful for a programming trick.

When you do a comparison like `TRY = 2`, BASIC actually returns a number. If TRY actually is 2, BASIC returns the value 1; if TRY is anything else, BASIC returns 0. The IF statement actually takes a number for the condition. If the number is 0, the IF statement acts as if the condition were FALSE. If the number is anything except 0, the IF statement acts as if the condition is TRUE.

There's nothing magic about a comparison. Since BASIC handles comparisons and other Boolean operations using numbers, you can use a comparison anywhere you would use any other mathematical expression. The distinction is entirely in the way we think about Boolean operations, not anything internal to BASIC. Putting this to work, we can look at a Boolean value by printing it, like this:

```
PRINT 4 < 3
```

It's also possible to store Boolean values in a numeric variable. In fact, the tool interfaces GSoft BASIC loads automatically include a type called `BOOLEAN` and two constants, `TRUE` and `FALSE`. You can use these constants and the `BOOLEAN` type in all of your GSoft BASIC programs.

While comparisons are the most common Boolean operation, you'll encounter more complex expressions later in the course. The first of the Boolean operations is `AND`, which tests to see if two conditions are met instead of one. Here's an example that prints the values within a specific range.

```
FOR I = 1 TO 10
  IF ( I > 3 ) AND ( I < 8 ) THEN
    PRINT I
  END IF
NEXT
```


The AND operation is true if both of the conditions to either side are true, and false if either one of the conditions is false. In other words, it means the same thing in BASIC that it does in English when you use it as a condition. If you're not sure how AND works, type in this short program and watch it in action.

The second operation is OR. Like AND, it means the same thing in BASIC as it does in an English statement about conditions. If either condition is true, OR returns true; if both conditions are false, OR returns false. Here's a sample you can use to explore how this works.

```
FOR I = 1 TO 10
  IF ( I < 3 ) OR ( I > 8 ) THEN
    PRINT I
  END IF
NEXT
```

The last Boolean operation is NOT. Just as you'd expect, it reverses the meaning of a Boolean value. Here's a simple example you might use in the main part of a program that waits for the user to do something, then acts on whatever the user did. This is the way most desktop programs are organized, although you can organize any program that waits for user events this way.

```
DONE = FALSE
WHILE NOT DONE
  CALL DOEVENT
  CALL IDLEPROCESS
WEND
```


Lesson Four – Subroutines

Subroutines Avoid Repetition

In the first few lessons of this course all of the programs we are writing are fairly short. Many useful programs are short, but as you start to make your programs more sophisticated, the programs will get longer and longer. A simple game on the Apple IIGS, for example, is generally 1,000 to 3,000 lines long; most of the programs we have written so far are 20 to 60 lines long. As the size of your programs increase you will need some new concepts and tools to write the programs. One of the most important of these is the subroutine.

For our first look at subroutines, we will start with a program that draws three rectangles on the graphics screen, filling each with a different color.

```
REM Draw three colored rectangles with white outlines.

DIM I AS INTEGER :! Loop variable

! Set up for graphics.
HGR
SETPENMODE (0)

! Draw a white rectangle.
SETSOLIDPENPAT (15)
FOR I = 10 TO 120
  MOVETO (10, I)
  LINETO (250, I)
NEXT

! Draw a red rectangle.
SETSOLIDPENPAT (7)
FOR I = 61 TO 99
  MOVETO (220, I)
  LINETO (270, I)
NEXT
```

```

! Outline the red rectangle in white.
SETSOLIDPENPAT (15)
MOVETO (220, 60)
LINETO (220, 100)
LINETO (270, 100)
LINETO (270, 60)
LINETO (220, 60)

! Draw a blue rectangle.
SETSOLIDPENPAT (4)
FOR I = 81 TO 159
  MOVETO (50, I)
  LINETO (300, I)
NEXT

! Outline the blue rectangle in white.
SETSOLIDPENPAT (15)
MOVETO (50, 80)
LINETO (50, 160)
LINETO (300, 160)
LINETO (300, 80)
LINETO (50, 80)

! Wait for the user to press return.
INPUT " ";A$

```

If you look at this program closely you will see that there is very little difference between the parts that draw the red and blue rectangles. In fact, if we put the coordinates of the rectangles in variables called LEFT, RIGHT, TOP and BOTTOM, and put the color in a variable called COLOR, we could use exactly the same lines of code to draw the red and blue rectangles. The code would look like this:

```

! Draw a rectangle.
SETSOLIDPENPAT (COLOR)
FOR I = TOP + 1 TO BOTTOM - 1
  MOVETO (LEFT, I)
  LINETO (RIGHT, I)
NEXT

```

```

! Outline the rectangle in white.
SETSOLIDPENPAT (15)
MOVETO (LEFT, TOP)
LINETO (LEFT, BOTTOM)
LINETO (RIGHT, BOTTOM)
LINETO (RIGHT, TOP)
LINETO (LEFT, TOP)

```

While we don't really need to redraw the outline of the square for the white square, the same code could even be used to draw the white square. A few extra lines get executed when the outline is drawn (the outline is white, and so is the color that is filled in), but the same code could be used. One of the most common uses for a subroutine is just this situation. When your program needs to do essentially the same thing in several different places, you can write a subroutine to do the thing, and call it from more than one place. Let's try this in a program and then look at what is happening in detail.

```

REM Draw three colored rectangles with white outlines.

! Set up for graphics.
HGR
SETPENMODE (0)

! Draw white, red and blue rectangles.
CALL RECTANGLE(10, 250, 10, 120, 15)
CALL RECTANGLE(220, 270, 60, 100, 7)
CALL RECTANGLE(50, 300, 80, 160, 4)

! Wait for the user to press return.
INPUT ";"A$
END

!-----
!
! Rectangle - Draw a rectangle and outline it in white.
!
! Parameters:
!   left, right, top, bottom - edges of the rectangle
!   color - color of the inside of the rectangle
!
!-----

SUB RECTANGLE(LEFT AS INTEGER , RIGHT AS INTEGER , TOP AS
INTEGER , BOTTOM AS INTEGER , COLOR AS INTEGER )

```

```

DIM I AS INTEGER :! Loop variable

! Draw the rectangle.
SETSOLIDPENPAT (COLOR)
FOR I = TOP + 1 TO BOTTOM - 1
    MOVETO (LEFT, I)
    LINETO (RIGHT, I)
NEXT

! Outline the rectangle in white.
SETSOLIDPENPAT (15)
MOVETO (LEFT, TOP)
LINETO (LEFT, BOTTOM)
LINETO (RIGHT, BOTTOM)
LINETO (RIGHT, TOP)
LINETO (LEFT, TOP)
END SUB

```

The Structure of a Subroutine

The subroutine itself starts with the reserved word SUB. Right after the reserved word SUB is the name of the subroutine; ours is called RECTANGLE. You use this name in the rest of your program whenever you want to call the subroutine. “Calling” a subroutine is what programmers say when they mean that you want to execute the statements in the subroutine.

The stuff in parenthesis right after the subroutine name is the parameter list. In our subroutine the parameter list looks like this:

```

(LEFT AS INTEGER , RIGHT AS INTEGER , TOP AS INTEGER , BOTTOM
AS INTEGER , COLOR AS INTEGER )

```

It is no accident that this looks suspiciously like a DIM statement. In fact, if you remove the parenthesis and put DIM before the list you would have a perfectly legal DIM statement. What the parameter list actually does is define these variables within the subroutine. Any statement within the subroutine can use these variables. You can change them using an assignment statement, or use them in an expression, as we do in our program. A very important point to keep in mind, though, is that the variables actually go away after you leave the subroutine.

The SUB statement forms a sort of model that tells us how to call the subroutine, as you can see by comparing the SUB statement with a CALL statement from our sample program.

```
SUB RECTANGLE(LEFT AS INTEGER , RIGHT AS INTEGER , TOP AS
INTEGER , BOTTOM AS INTEGER , COLOR AS INTEGER )

CALL RECTANGLE(10,                250,                10,
                120,                15)
```

You see the SUB statement on the top line. The line is long enough that it's split across two lines in this book, but if you look at the program, you can see that it really is just one long line. The second line shows a call to the subroutine with spaces inserted to line up the matching fields. When a BASIC program starts to execute it is quickly scanned for SUB and FUNCTION statements, so when the CALL statement is executed, GSoft BASIC already knows that a subroutine named RECTANGLE is defined in the program, and that it expects 5 integer parameters. It expects them to appear after the procedure name, enclosed in parenthesis, and separated by commas. If you forget one of these parameters, put in too many, or use a parameter that can't be converted to an integer, the program will stop with an error.

When the subroutine is called BASIC starts by assigning the values you put in the parameter list of the CALL statement to the variables you defined in the parameter list of the SUB statement. In effect, for the call we are using as an example, BASIC does the following five assignments before the first statement of the subroutine is executed:

```
! In effect, this is what BASIC does.
LEFT = 10
RIGHT = 250
TOP = 10
BOTTOM = 60
COLOR = 0
```

When the subroutine starts, then, the variables from the parameter list already have an initial value.

After the parameters are set up the statements in the subroutine, like the statements in the program itself, are executed one after the other. This process continues until the END SUB statement is reached. At that point, control returns to the place where the CALL statement was issued, and execution picks up with the line right after the CALL statement.

In the RECTANGLE procedure you will find the variable I defined for use in a FOR loop. Like the parameters, the variables defined within the subroutine vanish after the subroutine finishes executing. The only thing you can access from the program is the subroutine itself—the variables and parameters don't even exist until after the CALL statement starts, and vanish before control returns to the CALL statement.

Where to Put Subroutines

When you run a program, GSoft BASIC starts by quickly scanning the program to locate all of the subroutines, then begins execution with the first line in the program. Since execution starts at the first line of the program, subroutines always appear at the end, after all of the lines in the program itself.

The order in which the subroutines appear doesn't really matter. I personally place them in alphabetical order to make it easier to find a particular subroutine, but that's just a habit I've formed over the years. About the only meaningful restriction is that each subroutine must be separate from all of the others—one subroutine's END SUB statement must appear before the SUB statement for the next one. The only lines that should appear between subroutines are comments or blank lines. And, of course, each subroutine's name must be different from the name of any other subroutine in your program.

The END Statement

As you know by now, right after the last line of a BASIC program executes the program stops. Now that you are using subroutines, though, you need to end the program a different way. The reason for this is that the SUB statement can be called, but not executed—so you need to force the program to stop before it starts executing the subroutines at the end of the program.

The END statement stops the program. You can use it anywhere, but you should always put an END statement right before the first subroutine.

Commenting Subroutines

A program with one subroutine isn't likely to be too confusing, but as our programs use more and more subroutines, there are some commenting conventions that will help make the programs easier to read.

In every programming language I use I always put a block of comments at the start of every subroutine. The exact format may vary from language to language to take advantage of specific features in the language, but there is no variation in the basic content. In each new computer language I learn, I quickly come up with a style that

works for that language and stick with it for all of my programs, making it much easier to move a subroutine from one program to a new program.

I use a very rigid format with up to five sections to comment a subroutine. If there is nothing to put in a section it's simply left out, which is why you only saw two sections in the RECTANGLE subroutine. Actually, two of the five sections deal with features you haven't seen yet. We'll get to them later in this lesson, but you're about to start writing subroutines of your own. Commenting your subroutines properly is an important habit to develop. It will save your hours and hours as your programs get longer. Because commenting is so important, we're going to look at the issues now so you can think about commenting from the very first subroutine you write.

Before looking at the sections in the block of comments, though, take a look at how the entire block of comments is set aside with a line of dashes. This gives an unmistakable visual cue, making it very easy to spot a block of comments as you scan the text in a program. The subroutine itself follows right after the block of comments. Once you've created your first subroutine, a quick copy and paste sets up these lines for all of the other subroutines you ever write—and I promise the effort will be worth it as your programs creep from dozens of lines to thousands.

Procedure Description

```
! Rectangle - Draw a rectangle and outline it in white.
```

The first thing in the block of comments tells what the procedure does. I generally try to keep this down to a single line, and never more than two lines. I put the name of the subroutine first, even though it appears right after the block of comments, because it's easier to find the comments when you're scanning text in the editor than it is to find the SUB statement, and with the name right at the top of the block of comments, I don't have to scan down the screen to find the subroutine name. It's a little thing, but it saves a lot of time.

There are a few cases where a single line doesn't adequately describe what a subroutine does. This doesn't happen as often as you might think, but it does happen. When a situation like this pops up, I still put a one line description next to the subroutine name, then I skip a line and give a detailed description. In effect, the first line is a title line, and the next lines expand on the title.

Parameters

```
! Parameters:  
!   left, right, top, bottom - edges of the rectangle  
!   color - color of the inside of the rectangle
```

Next is a parameter declaration section that describes the meaning of each parameter that appears in the SUB statement. Again, there is usually no need for more than a line, or perhaps two.

Shared Variables

Shared variables are variables from the main program that are used inside a subroutine. The section looks like the list of parameters, but it's labeled Shared variables rather than Parameters. You'll see this kind of comment later, when we start discussing shared variables in detail.

Return Values

There is a kind of subroutine called a function that returns a value. We'll see those later in this lesson. It's important to describe exactly what the function returns.

Notes

If the subroutine is based on some outside reference material, does something unexpected, or if there is anything I'd like to remind myself of if I ever need to come back and change the subroutine or move it to another program, I put the information in a notes section. The notes section looks something like this:

```
! Notes:
!   1. For a description of the insertion sort, see
!       "Algorithms + Data Structures = Programs," p. 85.
```

As with the other formatting and commenting conventions mentioned in this course, there are many correct ways to comment and format a subroutine that are different from the one I have shown you. The important point isn't which one you use; the important point is to find one you like that supplies the same information and use it consistently.

Subroutines Let You Create New Commands

We have seen that a subroutine can be used to take a series of similar, repetitious commands and place them in a single subroutine, making our program shorter and easier to understand. Subroutines can also be used to create new commands, which helps organize the program, making it easier to read. The RECTANGLE subroutine we have

already created is one example. Once you know what the RECTANGLE subroutine does, it is a lot easier to read the lines

```
! Draw white, red and blue rectangles.  
CALL RECTANGLE(10, 250, 10, 120, 15)  
CALL RECTANGLE(220, 270, 60, 100, 7)  
CALL RECTANGLE(50, 300, 80, 160, 4)
```

than it was to read the original program. The idea of using subroutines to neatly package our program is a very powerful one. It takes some getting used to, but once mastered, the technique will help you write programs faster and find errors in programs easier.

There is another advantage, too. Most people tend to write a few general types of programs. For example, an engineer might write several programs to deal with complicated matrix manipulation, but never deal with graphics to any great degree. Another person might use his computer to write adventure games. Any time you start writing programs that fall into broad groups like this, you will find that there are sections of your program that get repeated over and over again. By packaging these ideas into subroutines, you can quickly move the proper sections of code from one program to another.

As an example, let's look at a small section of code that seems to appear at the beginning of nearly all of our graphics programs.

```
! Set up for graphics.  
HGR  
SETPENMODE (0)  
SETSOLIDPENPAT (15)
```

We can package these three lines into a subroutine called INITGRAPHICS like this:

```
!-----  
!  
! InitGraphics - Set up for graphics  
!  
!-----  
  
SUB INITGRAPHICS  
HGR  
SETPENMODE (0)  
SETSOLIDPENPAT (15)  
END SUB
```

With this new procedure, our program becomes even easier to read:

```
! Set up for graphics.
CALL INITGRAPHICS

! Draw white, red and blue rectangles.
CALL RECTANGLE(10, 250, 10, 120, 15)
CALL RECTANGLE(220, 270, 60, 100, 7)
CALL RECTANGLE(50, 300, 80, 160, 4)

! Wait for the user to press return.
INPUT " ";A$
END
```

It may not be obvious yet, but there is still one more advantage to packaging even these three simple commands into a subroutine. At some point, you may decide that you want to set up the graphics screen a bit differently. For example, you may want to paint the entire screen white so the drawings appear on a white background rather than a black one. With the graphics initialization in a neat little package, it will be easy to redo the package and quickly update all of your programs. You will also learn faster ways to color in a rectangle. If all of your programs use the `RECTANGLE` subroutine, you can easily update the subroutine, quickly bringing all of your programs up to date. If the code to draw rectangles is scattered throughout your programs, though, it would be a daunting task to change them all, simply because it would be hard to find all of the places that need to be changed.

Problem 4.1. One use of the `RECTANGLE` procedure is to draw game boards. For example, a board for a Reversi game would consist of eight rows and eight columns of green squares with white outlines. A chess or checker board can be drawn as eight rows and eight columns of alternating black and white squares.

Use the `Rectangle` procedure to draw a checker board in the graphics window. Make each square 20 pixels wide and 20 pixels high, with the top left square at 5,5. Use colors of 5 and 12, which gives a gray and green board instead of the boring traditional black and white board.

Hint: Use one `FOR` loop nested within another to loop over the rows and columns, like this:

```
FOR ROW = 1 TO 8
  FOR COLUMN = 1 TO 8
    <draw a square>
  NEXT
NEXT
```

This way, you can locate the top of each square as $(\text{ROW} - 1) * 20 + 5$. The bottom of each square will be at $\text{ROW} * 20 + 35$. The same idea can be used to find the left and right edge of each square.

Functions are Subroutines that Return a Value

In the last lesson we used a pseudo-random number generator in several programs to create simulations. One common theme in these simulations was to restrict the range of the random number and force the single-precision result returned by the RND function into an INTEGER value. For example, in our number guessing game, we selected numbers from 1 to 100. To roll dice, on the other hand, we used the same idea to select a random number from 1 to 6. With what we have learned about subroutines it would seem that this would be an ideal candidate for packaging. There is a problem, though. The whole point of the random number code is to produce a number. We need a way to get a value back from the subroutine. When we need a value back, BASIC gives us a new flavor of the subroutine called a function. A function is just a subroutine that can return a single value.

Here's a program that demonstrates this idea by packaging our random number generator.

```
REM This program rolls two dice 20 times.

DIM SIDES AS INTEGER :! # of sides on the dice
DIM NUMDICE AS INTEGER :! # of dice to throw
DIM I AS INTEGER , J AS INTEGER :! loop/index variables
DIM VALUE AS INTEGER :! value rolled on a die

! Set up the number of dice and number of sides.
SIDES = 6
NUMDICE = 2
```

```

FOR I = 1 TO 20
  FOR J = 1 TO NUMDICE
    PRINT RANDOMVALUE(SIDES), ;
  NEXT
  PRINT
NEXT
END

```

```

!-----
!
! RandomValue - Return a random number in the range 1 to max
!
! Parameters:
!   max - maximum allowed value for the random number
!
! Returns: Random number in the range 1..max
!
!-----

```

```

FUNCTION RANDOMVALUE(MAX AS INTEGER ) AS INTEGER
DIM VALUE AS INTEGER :! Random value to return

VALUE = 1 + RND (1) * MAX
IF VALUE = MAX + 1 THEN
  VALUE = MAX
END IF
RANDOMVALUE = VALUE
END FUNCTION

```

There are really only two differences in the way you write a subroutine and function. The first shows up in the function header, which starts with the reserved word **FUNCTION**, rather than the reserved word **SUB**. The function returns a value. It is possible for this value to be an integer, a real number, or any other type we've covered so far in this course. Functions cannot return arrays or records, two types we'll cover later, but there are easy ways around that issue.

Naturally, you have to tell the compiler what type of value the function returns. You do this just like you would for a variable, by following the name of the function (and the parameter list, if there is one) with **AS** and the type. In the case of our **RANDOMVALUE** function, the type is **INTEGER**.

At some point you need to specify what value the function should return. This is the second difference between a function and a subroutine. Somewhere in the function, you

need to assign a value to the name of the function itself. You can do this in more than one place, if you like, using IF statements to determine which assignment decides the value of the function. You can also assign a value to the function more than once, perhaps starting it off with an initial value that may or may not get changed later. You must assign a value to the function at least one time, however. If you don't, the value returned by the function is zero or an empty string, but you really shouldn't count on this fact.

You can use a function anywhere you could use a value within the BASIC language. In our program, we use the function in the statement

```
PRINT RANDOMVALUE(SIDES), ;
```

When the program gets to this statement it calls the function. The function calculates a value and returns it. The value is printed, just as the number 4 would be in the statement

```
PRINT 4, ;
```

Problem 4.2. You can use a function anywhere you can use a value in BASIC. In particular, you can use the RANDOMVALUE function to decide how many times to loop through a FOR loop, like this:

```
FOR I = 1 TO RANDOMVALUE(20)
  ...
```

You can also use a function to set the value of a parameter for another subroutine or function call.

Use these ideas to create a program that will draw a random number of rectangles, not to exceed 30, in the graphics window. The rectangles should have a left and right value between 1 and 319, and a top and bottom value between 1 and 199. Use an IF statement and a temporary variable to make sure the left side is less than or equal to the right side, and that the top is less than or equal to the bottom, like this:

```
IF LEFT > RIGHT THEN
  TEMP = LEFT
  LEFT = RIGHT
  RIGHT = TEMP
END IF
```

Finally, the color of the rectangle should be chosen at random, and should be in the range 0 to 15. You can get a value from 0 to 3 from the RANDOMVALUE function like this:

```
RANDOMVALUE(16) - 1
```

The call to RANDOMVALUE to get the color of the rectangle should appear in the parameter list of the call to Rectangle.

Value and Variable Parameters

There are some places where we want to package some code that changes more than one value. A good example of this is the ball bouncing program from the last problem in Lesson 3. It would be nice to package the code that updates the position of the ball into a function and return the new position of the ball. There is a problem, though. A function can only return one value, but we need to update both an X and Y coordinate.

It turns out that there are two ways to pass a parameter in BASIC. If you pass a variable as the parameter, and not an expression, and if the variable is the same type as the subroutine is expecting for a parameter, any changes made in the subroutine are also made in the main program. Parameters passed this way are called variable parameters. If the parameter you pass is an expression of any kind at all, even something as simple as converting an INTEGER to a SINGLE, any changes made in the subroutine have no effect at all on the variable in the main program.

Let's look at some examples to see how this works. The first example passes a variable parameter.

```
DIM I AS INTEGER

I = 1
CALL TEST(I)
PRINT I
END

SUB TEST (J AS INTEGER)
J = J + 1
END SUB
```

Since the passed parameter I is the same type as the parameter variable J, and since there is no expression involved, I is passed as a variable parameter. This means that changing J in the subroutine changes the value in the main program, too, so the program prints 2.

With a very simple change, we turn the parameter into a value parameter.

```
I = 1
CALL TEST(I)
PRINT I
END

SUB TEST (J AS INTEGER)
J = J + 1
END SUB
```

The only change was to drop the DIM statement that declared I as an INTEGER. As a result, I is defined with the default type of SINGLE, the type for a single-precision floating-point variable. This program prints 1, because the change to J in the subroutine does not change the value of I in the main program.

This brings up a dirty little problem in the BASIC programming language. Every language has features that sometimes cause problems; this is one of them for BASIC. The problem is that it's easy to change a parameter inside a subroutine, then have the subroutine change the value of a variable in the main program by accident. This isn't an obvious bug. You may end up scratching your head for quite a while before you finally discover the problem. There is one defensive programming technique I would recommend in all BASIC subroutines that are not supposed to change the value of a parameter, and that is to use a separate variable internally if you need to change a parameter value. In our simplistic example, the change would look like this:

```
DIM I AS INTEGER

I = 1
CALL TEST(I)
PRINT I
END

SUB TEST (J AS INTEGER)
DIM K AS INTEGER

K = J
K = K + 1
END SUB
```

In this example, I is not changed in the main program, even though it is passed as a variable parameter. Another way to protect a value from the part of the program making

the call is to turn the parameter into an expression. The traditional way to do this in BASIC is to put parenthesis around the parameter, like this:

```
DIM I AS INTEGER

I = 1
CALL TEST((I))
PRINT I
END

SUB TEST (J AS INTEGER)
J = J + 1
END SUB
```

Once again, this simple change is enough to change the parameter from a variable parameter into a value parameter, and the program prints 1. In general, though, the extra typing is a bit of a pain, and you'll quickly stop using the parenthesis unless you know they are needed. That's why I prefer writing the subroutine so it won't change the value of a parameter unless that's the point of the subroutine.

And finally, here's just such an example. This is my solution to Problem 3.4 rewritten using subroutines.

```
REM Draw a ball bouncing across the screen.

DIM X AS INTEGER , Y AS INTEGER :! Coordinates for the ball
DIM XSPEED AS INTEGER , YSPEED AS INTEGER :! Speed of the ball
DIM ITER AS INTEGER :! Number of iterations
DIM I AS INTEGER :! loop counter

! Get the ball's initial position, speed, and the number
! of animated frames.
INPUT "Starting X position: ";X
INPUT "Starting Y position: ";Y
INPUT "X Speed          : ";XSPEED
INPUT "Y Speed          : ";YSPEED
INPUT "Number of steps  : ";ITER

! Set up the graphics window.
CALL INITGRAPHICS
SETPENMODE (2)
SETPENSIZE (4, 4)
```

```
! Make sure the starting position is on the screen.
CALL RESTRICT(X, 0, 319)
CALL RESTRICT(Y, 0, 199)

! Animate the ball.
MOVETO (X, Y)
LINETO (X, Y)
FOR I = 1 TO ITER
    CALL MOVEBALL(X, Y, XSPEED, YSPEED)
NEXT
INPUT ";A$
END

!-----
!
! InitGraphics - Set up for graphics
!
!-----

SUB INITGRAPHICS
HGR
SETPENMODE (0)
SETSOLIDPENPAT (15)
END SUB

!-----
!
! MoveBall - move the ball
!
! Move a ball in the graphics window.  If the ball hits one
! of the sides, the direction of the ball is changed.
!
! Parameters:
!   X, Y - position of the ball
!   VX, VY - velocity of the ball
!
!-----

SUB MOVEBALL(X AS INTEGER , Y AS INTEGER , VX AS INTEGER , VY
AS INTEGER )

DIM X2 AS INTEGER , Y2 AS INTEGER :! New position for the ball
```

```
! Find the new X position for the ball
X2 = X + VX
IF X2 < 0 THEN
    X2 = 0
    VX = - VX
ELSE IF X2 > 319 THEN
    X2 = 319
    VX = - VX
END IF
```

```
! Find the new Y position for the ball
Y2 = Y + VY
IF Y2 < 0 THEN
    Y2 = 0
    VY = - VY
ELSE IF Y2 > 199 THEN
    Y2 = 199
    VY = - VY
END IF
```

```
! Draw the ball at the new position.
MOVETO (X2, Y2)
LINETO (X2, Y2)
```

```
! Erase the old ball.
MOVETO (X, Y)
LINETO (X, Y)
```

```
! Update the ball position.
X = X2
Y = Y2
END SUB
```

```
!-----
!  
! Restrict - make sure a value is inside a given range  
!  
! Parameters:  
!   X - value to restrict to a range  
!   LOW, HIGH - allowed range of values  
!  
!-----
```

```
SUB RESTRICT(X AS INTEGER , LOW AS INTEGER , HIGH AS INTEGER )
IF X < LOW THEN
  X = LOW
ELSE IF X > HIGH THEN
  X = HIGH
END IF
END SUB
```

In this program the MOVEBALL subroutine is used to update the position of the ball on the screen. We pass four values to the MOVEBALL subroutine; the current x and y position of the ball and the current velocity of the ball. Each of these four variables can be changed by the subroutine.

Problem 4.3. By using our neatly packaged subroutine you can quickly write a program to bounce more than one ball around on the screen. Modify the sample program to bounce 10 balls simultaneously.

Use the RANDOMVALUE function to choose the initial positions and speeds of the balls. Move the balls 100 times.

Shared Variables

With the exception of parameters, any variable declared in the program can't be used from inside a subroutine or function, and any variable declared inside a subroutine or function can't be used from the main program. Fortunately, there is a way to change all that so variables other than parameters can be shared among the various parts of the program. Cleverly enough, it's done with the SHARED command.

The SHARED command is used inside a subroutine when it needs to use a variable declared in the program. The command is pretty simple; you just put the name of the variable right after the word SHARED. If you want to share several variables with a single SHARED command, list all of the variables separated by commas.

One good way to use SHARED variables is to set up values used throughout a program. For example, our graphics programs frequently use the number of pixels on the screen as a boundary, making sure balls bounce off the edge and so forth. This boundary value can change. If you move on to toolbox programming you'll discover that there is another way to draw on the Apple IIGS that uses 640 horizontal pixels rather than 320. You'll also learn to create windows, and these windows are generally smaller than the physical screen. By placing the screen size in shared variables you can use a single value throughout the program, making it easy to change the screen size if you use the same subroutine in a later program.

Here's a short example that shows how to use shared variables. As our programs get longer and more complicated, we'll find many uses for them that aren't so simple!

```
REM Draw a big X across the graphics screen.

DIM MAXX AS INTEGER , MAXY AS INTEGER :! Size of the graphics
screen

! Initialize the size of the graphics screen.
MAXX = 320
MAXY = 200

! Initialize the graphics screen.
CALL INITGRAPHICS

! Draw a big x across the screen.
CALL X

! Wait for the user to press return.
INPUT "";A$
END

!-----
!  
! InitGraphics - Set up for graphics  
!  
!-----

SUB INITGRAPHICS
HGR
SETPENMODE (0)
SETSOLIDPENPAT (15)
END SUB

!-----
!  
! X - Draw a big X across the screen  
!  
! Shared variables:  
!   maxx, maxy - size of the screen  
!  
!-----
```

```
SUB X
  SHARED MAXX, MAXY

  MOVETO (0, 0)
  LINETO (MAXX, MAXY)
  MOVETO (0, MAXY)
  LINETO (MAXX, 0)
END SUB
```


Lesson Five –Strings

What Are Strings?

You may have noticed that a string was the first data type we ever dealt with, but you haven't seen much of them. Back in Lesson 1 our very first program wrote a string constant to the screen. Since then we have made extensive use of integers and real numbers, but the only string variables we've used were on INPUT statements, and with the exception of a very brief aside, that was really just a way to wait for the user to press return before ending a program. In this lesson we'll delve deeper into the mysteries of strings, learning how to declare them and how to manipulate strings in our programs.

In BASIC a string is a simple variable, just like in integer or a real number. Unlike a number, though, a string does not have a fixed length. It can vary from no characters at all to a whopper of a string with 32767 characters. It's important to keep in mind that the upper limit on the length of a string varies from one implementation of BASIC to another. In most implementations the upper limit is 255 characters, and frankly, that's enough for most situations.

The characters in a string are any of the ASCII characters. The ASCII characters are the 95 printing characters you see on your keyboard and 33 special purpose characters, some of which, like the return key, are also on your keyboard. There's a chart of them a little later in the lesson, in the section where we discuss character values in detail. GSoft BASIC also allows the extended ASCII characters supported by Apple on the Apple IIGS and Macintosh lines of computers.

Since strings can vary in length, they aren't stored the same way as numbers. A string variable actually contains information about the location of the characters. The characters in the string are stored in a separate area of memory in your computer. This puts some limitations on what we can do with string values. We'll talk about them in more detail as various topics come up in the course, but in a nutshell, you can't fake the creation of a string or change the length of an existing string behind BASIC's back—you must allow BASIC to create, change, and delete strings.

The Two Ways To Read a String

You got a brief look at reading strings from the keyboard in Lesson 3, when we used this short program to show that the INPUT statement could read strings, too, and not just numbers.

```
INPUT "Please type your name: "; NAME$
PRINT "Hello, "; NAME$
```

We didn't delve deeper at that time, but now it's time to look at a major weakness in the INPUT statement for reading strings. It's obvious when you make a slight change in the program, like this:

```
INPUT "Please type your city, state and zip code: "; ADDRESS2$
PRINT ADDRESS2$
```

The natural thing to type (in the United States, anyway) is something like this:

```
Albuquerque, New Mexico 87120
```

Try it. Everything after the comma is lost. The fact is, the INPUT statement just doesn't handle commas well when reading strings. Just as with a number, the comma signals the end of the string.

That's not always bad. In fact, in this particular case, it can be very useful. Let's try the program again:

```
DIM CITY AS STRING , STATE AS STRING , ZIP AS LONG

INPUT "Please type your city, state and zip code: ";CITY,
STATE, ZIP
PRINT CITY, STATE, ZIP
```

If you remember to put a comma after the state, like this:

```
Albuquerque, New Mexico, 87120
```

BASIC divides the typed text neatly into two strings and a number, storing the results in appropriate variables.

But more often than not, experienced BASIC programmers find that the way INPUT handles commas is more of a hindrance than a help. That's why there is another form of the INPUT statement in BASIC called LINE INPUT. The LINE INPUT statement looks just like the INPUT statement. In fact, there is really only one difference: Instead of separating the various values you type with commas, you must put them on a separate line. In most cases you'll end up using one LINE INPUT statement for each line.

Here's the last version of our program for reading the city, state and zip code. This one reads the entire line, commas and all, into a string variable.

```
LINE INPUT "Please type your city, state and zip code: ";
ADDRESS2$
PRINT ADDRESS2$
```

Manipulating Strings

BASIC only has five operations for manipulating strings, but surprisingly, they are enough for any task you'd like to perform. You can easily create more specialized operations based on the ones BASIC already has.

The simplest of all of the operations is technically known as string concatenation. That's just a fancy term for attaching one string to the end of another. BASIC uses the + operation to concatenate strings, which sort of makes sense, because you're adding one string to the end of another. For example,

```
A$ = "test"
B$ = "ing"
PRINT A$ + B$
```

prints

```
testing
```

String concatenation gives you an easy way to combine strings to form a bigger one; the next three functions give you a way to extract a piece of a long string. LEFT\$ and RIGHT\$ pull characters from the left or right end of a string. Each of these functions takes two parameters, a string and the number of characters you want. It is legal to ask for more characters than are actually in the string; if you do that, you will get the entire string back. You can see how this works by running this short program, which peels characters off of the left edge of a test string.

```
A$ = "testing"
FOR I% = 0 TO 8
  PRINT I%, LEFT$ (A$, I%)
NEXT
```

The last of the most fundamental string operations is LEN, which figures out how many characters are in a string. LEN takes a single parameter, a string, and returns the number of characters in that string.

Let's put these statements to work in a real program. This particular program takes a string and reverses the order of the characters. It's a cute gag program, but it also shows clearly how LEFT\$, RIGHT\$ and concatenation can be used to tear apart a string and put it back together in a wholly different way. Just as important, it shows how to package this operation as a BASIC FUNCTION, in effect creating a new string manipulation command that you can copy from one program and paste into others that need to do the same operation.

```
REM Reverse
REM
REM This program reads in a string, reverses the order of the
REM characters, and writes the string back to the text screen.
REM It continues doing this until a string of length zero is
REM entered. To get a string of length zero, press the RETURN
REM key without typing any other character.

DIM INSTRING AS STRING :! input string
DIM OUTSTRING AS STRING :! output string

! Loop until there is no input string.
DO
  ! Get a string.
  LINE INPUT "String to reverse: ";INSTRING

  ! Reverse the characters in the string.
  OUTSTRING = REVERSE$(INSTRING)

  PRINT "Reversed string : ";OUTSTRING
  PRINT
LOOP WHILE LEN (INSTRING) <> 0
END
```

```

!-----
!
! Reverse$ - Reverse the characters in a string
!
! Parameters:
!   s - string to reverse
!
! Returns: String with the characters reversed
!
!-----
FUNCTION REVERSE$(S AS STRING ) AS STRING

DIM I AS INTEGER :! loop variable
DIM S1 AS STRING :! remaining characters in the input string
DIM S2 AS STRING :! string with characters reversed

S1 = S
S2 = ""
FOR I = 1 TO LEN (S1)
    S2 = S2 + RIGHT$ (S1, 1)
    S1 = LEFT$ (S1, LEN (S1) - 1)
NEXT
REVERSE$ = S2
END FUNCTION

```

It may seem strange to create a completely new string variable, S1, to hold the same string that was passed as a parameter. If you think so, try taking it out and renaming the parameter S1. What happens to the program?

I did warn you about this sort of thing. In the last lesson, I suggested that you always copy a parameter into a local variable if you would be changing the value of the variable in the subroutine, since it was possible you would change the variable in the original program, with unwanted results. That's just what happens in this case if you don't make a copy of the original parameter. The REVERSE\$ function gradually removes characters from one string while building a second. When it finishes, the original string has been reduced to an empty string, which is another name for a string that doesn't have any characters. Back in the main program the DO loop ends unexpectedly because INSTRING got changed to the empty string, too.

The last of the five string manipulation functions is MID\$. This function takes characters from the middle of the string rather than the right or left side. MID\$ uses three parameters rather than two. The first is still the string to work on. Next comes the index of the first character you want MID\$ to return, counting from 1. The last parameter is the number of characters you want back. For example,

```
PRINT MID$("This is a test.", 6, 2)
```

prints the second word, “is”. Remember, spaces are characters, too, so the space between “This” and “is” counts as a character.

Like LEFT\$ and RIGHT\$, MID\$ does sensible things if you ask for characters that are not there. If you start in the middle of the string and ask for more characters than there are left in the string, MID\$ returns the ones that are there. For example,

```
PRINT MID$("This is a test.", 11, 50)
```

returns the string “test.” If the index is larger than the number of characters in the string, MID\$ returns an empty string.

Here’s the string reversing program, rewritten to use MID\$ and a backwards-stepping FOR loop to reverse the characters in the input string.

```
REM Reverse
REM
REM This program reads in a string, reverses the order of the
REM characters, and writes the string back to the text screen.
REM It continues doing this until a string of length zero is
REM entered. To get a string of length zero, press the RETURN
REM key without typing any other character.

DIM INSTRING AS STRING :! input string
DIM OUTSTRING AS STRING :! output string

! Loop until there is no input string.
DO
  ! Get a string.
  LINE INPUT "String to reverse: ";INSTRING

  ! Reverse the characters in the string.
  OUTSTRING = REVERSE$(INSTRING)

  PRINT "Reversed string : ";OUTSTRING
  PRINT
LOOP WHILE LEN (INSTRING) <> 0
END
```

```

!-----
!
! Reverse$ - Reverse the characters in a string
!
! Parameters:
!   s1 - string to reverse
!
! Returns: String with the characters reversed
!
!-----
FUNCTION REVERSE$(S1 AS STRING ) AS STRING

DIM I AS INTEGER :! loop variable
DIM S2 AS STRING :! string with characters reversed

S2 = ""
IF LEN (S1) > 0 THEN
  FOR I = LEN (S1) TO 1 STEP - 1
    S2 = S2 + MID$ (S1, I, 1)
  NEXT
END IF
REVERSE$ = S2
END FUNCTION

```

If you looked closely at this example, you may have noticed a feature of the FOR loop we've never covered before. The STEP size of -1 is used to tell the FOR loop to loop from a large number down to a small one. For example,

```

FOR I = 10 TO 1 STEP -1
  PRINT I
NEXT

```

does a countdown from 10 to 1. Other than counting down instead of up, this loop works just like all of the other FOR loops you've used.

In some ways this version is simpler than the one that uses LEFT\$ and RIGHT\$, and in some ways it is more complex. This version doesn't need to make a copy of the input parameter, since it isn't changed, and it uses one fewer statements inside the FOR loop to manipulate the strings, since it doesn't have to remove a character from the input string. On the other hand, it needs an extra IF statement to make sure there are characters in the string before the FOR loop starts.

As a general rule, this version is better than the first. They are both about the same size and complexity, but the version based on MID\$ has one fewer statement in the FOR

loop. While there are certainly exceptions, programs are generally faster when you reduce the number of statements inside a loop. That's because statements in a loop are almost always executed more times than statements that are outside of the loop, sometimes thousands of times more often, so moving things out of the loop tends to make the program faster. In many programs, the difference is minimal or unimportant, but in others the difference is dramatic. In fact, reducing the number of operations in a loop is one of the most effective ways to make a slow program run faster.

Problem 5.1. If you do a lot of string manipulations, you'll start to build up a library of more powerful commands. One that you might add is `INSERT$`, which inserts a string in the middle of an existing string.

Write a function that takes two strings and a position as input. Your function should insert the second string parameter into the first at the position given by the third, numeric parameter. Be sure your subroutine handles any argument reasonably. If the position is less than 1, the second string should appear at the beginning of the first string. If the position is greater than the length of the first string, the second string should appear at the end of the first string.

Write a program that tests this function. It should use a sequence of input statements to read test strings and a position, then print the result returned by the function. The program should stop if you press return immediately for both input strings, but not until it calls `INSERT$` with this odd case!

Test your program with every combination of add data you can think of. Do your best to trick your subroutine, trying to make it fail. If it's going to fail, it's best if it fails while *you* are testing it, rather than later, when someone is using your program!

Characters

Way back in Lesson 1, you learned that

```
"Hello, world."
```

is a string constant. A string constant consists of any number characters enclosed in quote marks. That "any number" is quite literal—two quote marks in a row form a legal string constant for a string with no characters. In various books you'll see this called the empty string or the null string. Any character you can type can appear in a string constant except for the double quote mark itself. Later in the lesson we'll find a way to force the quote mark into a string.

The ASCII Character Set

Characters and integers enjoy a special relationship with each other. To decide what it means to compare two strings, for example, we need to decide if one character is less than another. While you can get pretty good agreement from most people whether the character 'a' is less than the character 'b', things get a little less definite when you ask if the character '^' is less than the character '*'. For this reason, as well as other reasons we'll explore, we often convert characters to integers and integers to characters.

There are two functions in BASIC that are used to convert characters to numbers and numbers to characters. The CHR\$ function takes a number as a parameter and returns a string with a single character. The number should be in the range 0 to 255; if it is not, CHR\$ adds or subtracts 256 from the value you give until the number is in this range.

The ASC function does just the opposite. It takes a string and returns the numeric value associated with the first character in the string. If the string has no characters, ASC returns the value 0.

The ASCII character set defines the relationship between the characters and their numeric equivalents. It also lists all of the characters you can use. It has one character for each of the values from 0 to 127. Some of these values are known as printing characters. For example, the numeric value 65 is used to represent an uppercase 'A'. The lowercase letter 'a' is represented by 97. Some of the values in the ASCII character set are non-printing characters. These are used for special purposes. The character whose value is 13, for example, is used to separate lines in files of characters and to move to a new line on the text screen.

The table below shows the complete ASCII character set in tabular form. Non-printing characters are shown as the name of the value. To obtain the integer value used to represent one of the characters, add the number at the top of the column to the number at the start of the row. Try that for 'A' and 'a', which have values of 65 and 97, to make sure you understand how this works.

	0	16	32	48	64	80	96	112
0	nul	dle		0	@	P	`	p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	\$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
8	bs	can	(8	H	X	h	x
9	ht	em)	9	I	Y	i	y
10	lf	sub	*	:	J	Z	j	z
11	vt	esc	+	;	K	[k	{
12	ff	fs	,	<	L	\	l	
13	cr	gs	-	=	M]	m	}
14	co	rs	.	>	N	^	n	~
15	si	us	/	?	O	_	o	rub

The ASCII character set is the dominant character set on microcomputers, but it is not universal. On the Apple IIGS, and on most microcomputers, you can write your programs specifically for the ASCII character set. If you will be writing programs that must run on a variety of computers, though, you should be aware that the numeric equivalents of characters may vary. If possible, find out what character set is used on the various machines before you start to write your program, and make sure it will work with all of the character sets.

Problem 5.2. Write a program that loops over the numbers from 32 to 126, converts these numbers to strings using the CHR\$ function, and prints the characters to the screen. Skip to a new line after every 16 characters.

Modify this program to switch to the graphics screen. Use the MOVETO command to move to 15, 15 before you start to print the characters. As you can see, you have a simple but effective way to put text on the graphics screen.

The Extended Character Set

Apple defined extensions to the ASCII character set to allow Macintosh and Apple IIGS computers to display special characters used in non-English speaking countries that still use more or less the same alphabet as English speaking countries. This extended character set is not implemented for the text screen that most of our programs use, but it is available on the graphics screen. The characters in the Apple extended character set are shown in this table.

	00	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
0			0	@	P	`	p	Ä	ê	†	∞	¿	-			
1		!	1	A	Q	a	q	Å	ë	°	±	;	-			
2		"	2	B	R	b	r	Ç	í	¢	≤	¬	"			
3		#	3	C	S	c	s	É	î	£	≥	√	"			
4		\$	4	D	T	d	t	Ñ	î	§	¥	f	'			
5		%	5	E	U	e	u	Ö	ï	•	μ	≈	'			
6		&	6	F	V	f	v	Û	ñ	¶	∂	Δ	÷			
7		'	7	G	W	g	w	á	ó	ß	Σ	«	◇			
8		(8	H	X	h	x	à	ò	©	Π	»	ÿ			
9)	9	I	Y	i	y	â	ô	©	π	...				
A		*	:	J	Z	j	z	ä	ö	™	∫	spc				
B		+	;	K	[k	{	ã	õ	'	ª	À				
C		,	<	L	\	l		â	ú	¨	º	Ã				
D		-	=	M]	m	}	ç	ù	≠	Ω	Õ				
E		.	>	N	^	n	~	é	û	Æ	æ	Œ				
F		/	?	O	_	o		è	ü	Ø	ø	œ				

- The characters from the space (\$20) to the tilde (\$7E) are all standard printing ASCII characters.
- While they have standard definitions, the characters \$11..\$14, \$AD, \$B0..\$B3, \$B5..\$BA, \$BD, \$C2..\$C6 and \$D6 tend to be rare in most fonts.
- Character \$CA is the non-breaking space.

One thing worth pointing out is that you can change the font used to draw characters when you are using the graphics screen. The reason this is important is that not all fonts implement all of the special characters you see in the table. If one of the characters shown exists in the font, it will almost always use the character code shown, but there aren't many fonts that implement all of these characters. In fact, many specialized fonts don't implement any of the characters you see in the table, even the standard ASCII characters. For example, there are Hebrew, Greek, and hieroglyphic fonts, not to mention symbol fonts that implement all sorts of pictures as font characters. You generally have to try the fonts to see what they actually do.

Problem 5.3. Write a program that displays all 256 possible characters on the graphics screen. Some characters won't exist. To account for this fact, try drawing each character at a screen coordinate that matches the position of the character with a position in the table of fonts.

You can do this by using a MOVETO command right before you draw each character. One odd fact you must take into account is that the position you move to specifies the base line for the character. This is the bottom left position for a character

like M that lies entirely above the baseline. For a character like y, this position is still at the left edge of the character, but it is part way up, roughly where the tail starts to dip below the rest of the character. You will need to experiment a bit to find the proper number of pixels to leave between each character.

While the program will work either way, you will be able to see the characters easier if you paint the screen white with a series of MOVETO and LINETO commands before drawing the characters themselves.

P-Strings, C-Strings, and Other Confusions

If you have read much about programming or browsed through Apple's toolbox reference manuals, you know that there are several kinds of strings in the various computer languages. The two most popular are generally called p-strings and c-strings. The toolbox manuals also refer to text blocks, and the Apple IIGS disk operating system makes use of still another format for encoding file names.

P-strings get their name from the Pascal programming language because microcomputer implementations of Pascal like UCSD Pascal popularized the format. Ironically, they have little to do with the official definition of Pascal, but that's another story! The first character position in a p-string is actually occupied by a number, not a character. This number is the number of characters that follow. On almost all computers, and certainly on all of the computers where I've seen p-strings used, each character uses one byte of storage. While we won't go into the details of representing numbers using bits and bytes, take my word for it that this means the number of characters in a p-string is limited to the range 0 to 255 on practically any computer. The characters in the string itself follow right after the length byte.

C-strings are named after the C programming language, which is the most famous language that uses them. C strings are a sequence of characters followed by a null terminator, which is a character whose numeric value is zero. This gives another common name for this kind of string, the null terminated string.

As you can see, one advantage of c-strings over p-strings is that there is no fixed limit to the number of characters in a single string. A minor disadvantage is that you have to scan the entire sequence of characters looking for the null terminator to find the length of a string, which is a very common operation when you are doing string operations.

Text blocks are just sequences of characters. You can't tell from looking at the string itself how many characters it has; the length is kept in a separate variable.

GS/OS, the Apple IIGS disk operating system, uses something a lot like a p-string, but instead of using a single byte to represent the length of the string GS/OS uses two bytes. This gives a theoretical upper limit of 65535 characters in a string. GS/OS itself limits the size of a path name to 8192 characters, but there is nothing to prevent a future version

from implementing a larger limit. GS/OS also uses a variation on this format that has two lengths rather than one. The first value is the amount of memory available for the string, while the second is the current length.

With all of these formats available, it's fair to ask what GSoft BASIC uses. Essentially, GSoft BASIC uses c-strings. The format consists of a sequence of characters followed by a null terminator. There are some internal limits in the microprocessor used in the Apple IIGS that make it easier to deal with strings that are no longer than 32767 characters, though, so GSoft BASIC imposes an upper limit of 32767 characters on each individual string. If you try to create a string longer than 32767 characters, it is truncated. The final string is made up of the first 32767 characters of the string you would expect if there was no upper limit.

Comparing Strings

The same comparison operations used with numbers can also be used with strings. Two strings are compared by comparing the characters in the string, one after the other, until the characters don't match. One string is "less than" another if the numeric value of the first nonmatching character is less than the numeric value of the character at the same position in the second string.

For example, "A" is less than "B", since the numeric value of the character "A" is 65, while the numeric value of the character "B" is 66. Following the rules, "that" is less than "this", since the numeric value of the first nonmatching character, the "a" in "that", is 97, while the numeric value of the "i" from "this" is 105.

If two strings are not equal in length, but all of the characters up to the end of the shorter string match, then the shorter string is less than the longer one. And, of course, if the strings are the same, they are equal.

Looking at the ASCII character chart and thinking about these rules, they seem to make a lot of sense. Words compare pretty much the way we would expect from looking in, say, a dictionary. If a word is alphabetically before another, BASIC will say the first word in alphabetical order is less than the second. There is one major exception, though. Comparing strings fails to match our preconceptions miserably if one of the strings uses uppercase letters but the other does not. In the ASCII character chart, uppercase letters always come before lowercase letters, so the string "Washington" is less than the string "president". You can take care of this problem by converting both strings to all uppercase letters or all lowercase letters before comparing them, though.

Problem 5.4. Write a function and a program to test it that converts any string to all uppercase characters. Make sure the function does not change characters that are not lowercase alphabetic characters.

Numbers and Strings

The two remaining string functions take care of a chore that is pretty tough to do by writing your own subroutines: Converting strings to numbers and numbers to strings.

The STR\$ function takes any number and converts it to a string that has the same characters you would see if you used the PRINT statement to print the same number. It's cousin, the VAL function, takes a string and converts it to a numeric value. VAL always returns a double-precision floating-point number; STR\$ can take any number format, and formats each according to the rules used for that type of number by PRINT.

One use for these functions is in programs that need foolproof input of numbers. As you know by now, an INPUT statement that expects a number will ask for one if the user of the program enters something the program can't handle, but the way it handles the error may not be exactly what you're after. You can use the LINE INPUT statement, though, and read the input as a string. It's not all that tough to write a subroutine that will check to see if the text is a valid floating-point number, and it's pretty easy to check to see if the text is an integer. If your subroutine reports that the text is a number, you can convert the value easily with VAL. If the text the user typed is not a number, you can handle the error in a way that is more appropriate in your program than the severe method BASIC uses by default.

Garbage Collection

As your programs get longer, and especially if they use lots of strings, you may occasionally notice a slight pause. This is probably garbage collection. We'll explore what garbage collection really is and how you can manage it in this section.

Each time you create a string in BASIC it is stored in an area of memory that BASIC sets aside for variables. You can visualize the process as writing the string on a line of notebook paper. When the next string is formed, it is written on the next line. This process continues until memory fills up completely.

To see how this works, let's follow a very simple program,

```
A$ = "Test 1"  
B$ = "Test 2"  
A$ = "Test 3"
```

Following along on paper, the lines on the notebook paper version of memory look like this:

```
Test 1
```

```
Test 2  
Test 3
```

You might object that A\$ contains the string “Test 3” when the program finishes, so the string “Test 1” is no longer needed. You’re right, but it’s still in memory. It’s garbage, and the process of garbage collection is nothing less and nothing more than checking all of the variables in the program to see which ones are string variables, and of those, which strings they are actually using, then deleting the strings that are no longer needed. When garbage collection finishes, the strings in memory would be

```
Test 2  
Test 3
```

The problem is that checking all of the variables in your program and compressing the memory can take a noticeable amount of time. Most of the time it’s not noticeable, and even when it is it’s not worth worrying about, but every once in a while you will write a program that is just plain annoying to use if garbage collection happens at a particular point while the program runs. Maybe that’s right in the middle of an animation, or during a time-critical part of a communications program. Whatever the reason, you can force BASIC to do garbage collection using the FRE function. This forces BASIC to do garbage collection, which makes it far less likely that garbage collection will happen in the next few lines of code.

The FRE function takes a parameter. For garbage collection, it should be 0. It also returns a value. The value returned is the number of free bytes that are left in the variables area. That’s a good way to see if you’re running out of memory, which could cause garbage collection to occur way too often, slowing the program down a lot. If you have less than 10000 bytes of free space, I’d suggest you should increase the amount of memory. We won’t cover how that’s done in this course, but you can find the appropriate commands in the GSoft BASIC reference manual.

Don’t overuse the FRE function! Even if there is little or no garbage collection to do, the FRE command can take a fair amount of time. If you use it too often the entire program can slow noticeably. In fact, you should not use the FRE command at all unless you are trying to control when the garbage collection is done. BASIC will do garbage collection automatically whenever it is needed, and your program will run fastest if you let BASIC choose when to do garbage collection. The only advantage to FRE is that you can force the garbage collection to occur before a time-critical section of the program starts to execute.

Lesson Six –Arrays

Groups of Numbers as Arrays

Computers can deal with very large amounts of data. On the Apple IIGS, you can easily write programs that will deal with thousands of numbers, names, zip codes, or whatever. So far, though, the methods we have for dealing with these values are fairly limited. A database of a hundred friends, each of whom has a name, street address, a city, a state, and a zip code would be a daunting task if each value had to be placed in a separate variable.

One way we have to deal with large amounts of data is called an array. An array is a group of values, each of which is the same type. We use an index to determine which of the values we want to access at a given time.

For our first look at an array, let's do a simulation of rolling dice. We've done this several times before, on a small scale, but this time we're going to roll the dice 10000 times and keep track of how many times we get a 2, how many times we get a 3, and so forth. We could, of course, use a separate variable for each of the totals, but that would get to be a bit tedious. Instead, we will use an array.

To define an array, you need to specify how many things you want in the array and what kind they are. In our case, we are adding up the number of times a particular value shows up on a pair of dice. We can get any value from 2 to 12 from a pair of dice, so the easy way to create the array is to use the numbers 2 to 12 as indexes. We'll define the array this way:

```
DIM TOTALS(12) AS INTEGER :! number of spots showing
```

With the array defined this way, we can put a number into the array or take one out by giving the name of the array followed by the index in parenthesis. For example, if the variable DICE contains the number of spots we rolled, the expression

```
TOTALS(DICE) = TOTALS(DICE) + 1
```

will take the current value from the array, add one, and store the changed value back into the array.

There is one subtle point here. The 12 as the index for the array says the last value in the array is indexed with 12, as in

```
PRINT TOTALS(12)
```

But what is the first value? Actually, for every array in BASIC, the index of the first value is 0. In our dice rolling program we will never use TOTALS(0) or TOTALS(1). In some programs wasting two integer numbers is a big deal. The space is important. In other programs, wasting a few bytes is not nearly as important as writing a program that is easy to understand. In this program we'll sacrifice the extra four bytes of space for clarity's sake.

As with regular variables you can specify what kind of value the array holds using the special type characters, so

```
DIM TOTALS%(12)
```

defines an array that works just as well in our program. As with our other programs, though, we'll usually dispense with the extra character by defining arrays with a named type in the DIM statement. One way isn't necessarily any better than the other. I generally use characters when I'm writing short programs, and use named types for longer ones.

One other interesting feature about BASIC is that you can have an array and a variable with the same name. This is usually something you find out when you make a mistake and start trying to find out why a program doesn't work! I wouldn't recommend using the same name for a variable and an array because it's easy to get the two confused.

You can use an element of the TOTALS array anywhere that you could use an integer variable. You can, for example, print an element of an array, use it in an expression, or pass it as a parameter to a subroutine. There are very few cases, though, where you can use the entire array. You can't write an array using PRINT, for example. We will explore when and how you can use an entire array as we get to know arrays better.

Now, finally, it's time to look at a real program that uses arrays.

```
REM This program simulates rolling dice.  It counts the number  
REM of times each value appears, printing a summary after the  
REM run is complete.
```

```
DIM TOTALS(12) AS INTEGER :! number of spots showing
```

```
! Do the dice simulation.  
CALL SIMULATION(10000)
```

```
! Write the dice array.
CALL WRITEARRAY
END
```

```
!-----
!  
! RandomValue - Return a random number in the range 1 to max  
!  
! Parameters:  
!   max - maximum allowed value for the random number  
!  
! Returns: Random number in the range 1..max  
!  
!-----
```

```
FUNCTION RANDOMVALUE(MAX AS INTEGER ) AS INTEGER  
DIM VALUE AS INTEGER :! Random value to return
```

```
VALUE = 1 + RND (1) * MAX  
IF VALUE = MAX + 1 THEN  
    VALUE = MAX  
END IF  
RANDOMVALUE = VALUE  
END FUNCTION
```

```
!-----
!  
! Simulation - roll the dice, saving the results in totals  
!  
! Parameters:  
!   rolls - number of times to roll the dice  
!  
! Shared Variables:  
!   totals - array holding the total number of rolls  
!  
!-----
```

```
SUB SIMULATION(ROLLS AS INTEGER )
```

```
    SHARED TOTALS()
```

```

DIM I AS INTEGER :! loop variable
DIM SUM AS INTEGER :! # of spots for this roll

! Set the totals to zero.
FOR I = 2 TO 12
    TOTALS(I) = 0
NEXT

! Do the simulation.
FOR I = 1 TO ROLLS

    ! Roll the dice.
    SUM = RANDOMVALUE(6) + RANDOMVALUE(6)

    ! Increment the correct total.
    TOTALS(SUM) = TOTALS(SUM) + 1
NEXT
END SUB

!-----
!
! WriteArray - Write the results.
!
! Shared Variables:
!   totals - array holding the total number of rolls
!
!-----

SUB WRITEARRAY

SHARED TOTALS()

DIM I AS INTEGER :! loop variable

PRINT "spots", "times"
FOR I = 2 TO 12
    PRINT I, TOTALS(I)
NEXT
END SUB

```

Before you run this program, I want to let you know that it will take a long time. In fact, this program will run for over eight minutes on an accelerated Apple IIGS! This is the first computationally intense program you have seen in this course. If you like, you

can try various tactics to speed up the program. You can also use this program too see how big an impact sloppy coding might have. One easy example of this is using single-precision floating-point values instead of integer variables. If you switch all of the variables to real numbers the program actually takes over eleven minutes.

There is one other thing to notice about this sample program. In the last lesson you learned how to create shared variables so a value could be used in the main program and in a subroutine. This program shows how to use shared variables with an array. For the most part, sharing an array is done the same way as sharing a variable. If you remember, though, I said you could have an array and a variable with the same name, so you need some way to tell a variable from an array. You tell BASIC you want to share an array by placing the parenthesis after the array name. You don't put in the type or size of the array, though. Those values are adapted from the size and type declared in the main program.

Problem 6.1. There is often a trade-off between a program that is fast and a program that is easy to understand. Which factor is the most important is one that the programmer has to make as the program is written. The answer is really an engineering choice, and not something you can predict in advance.

The dice rolling program calls RANDOMINTEGER 20,000 times. That's really what takes most of the time. Change the program so it doesn't call a function by including the code from the RANDOMVALUE function inside of the FOR loop.

How much faster is the program?

The Shell Sort

There are a few basic tasks that show up over and over when you are writing real programs. One of these is sorting. If you use a program to keep track of your Christmas list, for example, you might want to sort the list by zip code so the Post office will let you send the Christmas cards out by bulk mail. If you want to check your Christmas list to see who's been naughty and nice, though, and are trying to find E. Scrooge, you may want the same list sorted alphabetically by name.

There are many ways to sort an array; each has its advantages and disadvantages. You will learn about other ways to sort an array later in the course, but we will start out now with one of the classic sorting methods. While there are faster ways to sort large arrays, the shell sort is very easy to understand, very easy to implement, and actually works better on short arrays than the more complicated sorts you will learn later.

The idea behind the shell sort is very simple. You start by scanning the array from front to back. At each step, you look to see if the value that comes after the current one in

the array is smaller than the current array element. If it is, you change them and continue scanning. As an example, we will sort the following array by hand.

<u>index</u>	<u>value</u>
1	6
2	43
3	1
4	6

We start off with the first array element and check to see if the value is smaller than the value in the second element of the array. (The arrow shows which element of the array we are working on.)

	<u>index</u>	<u>value</u>
à	1	6
	2	43
	3	1
	4	6

In this case, 6 is smaller than 43, so we do nothing. Moving on, we check the next element.

	<u>index</u>	<u>value</u>
	1	6
à	2	43
	3	1
	4	6

This time, 1 is smaller than 43, so we exchange the values in the second and third spots, ending up with this array:

	<u>index</u>	<u>value</u>
	1	6
à	2	1
	3	43
	4	6

Checking the third element, we find that 6 is also smaller than 43, so we again make a swap.

	<u>index</u>	<u>value</u>
	1	6
	2	1
à	3	6
	4	43

We don't check the last element of the array, since there is nothing that follows it.

At this point, we have successfully moved 43 to the last spot in the array, where it belongs, but the array is still not completely sorted. To sort the array completely, we need to keep track of whether or not we swapped any array entries. If we didn't need to swap any entries then the array is sorted. If we did swap two of the array elements, though, we need to make another pass over the array. Our second pass makes one swap, moving 1 to the first array element.

	<u>index</u>	<u>value</u>
	1	1
	2	6
	3	6
	4	43

Notice that we only want to swap elements of the array if the next element is actually less than the one we are inspecting. If we swap elements when the values are equal, we would loop over our sample array over and over, swapping 6 with itself on each pass.

Before diving into an example program that shows an actual sort, let's take a moment to examine concept that we will use that has nothing to do with arrays. It's something you saw briefly back in Lesson 3, but this is the first time it has appeared in a real program. While we are sorting the array, one of the things we need to keep track of is whether or not we have swapped any entries in the array. If we have, we need to make another pass through the array; if we have not swapped any entries, the sort is complete, and we can stop. One way to keep track of whether any swaps have been made would be to keep track of the number of swaps, and check to see if the number is zero. We could be a little more efficient, and set a number to zero, then set it to one if any swaps were made. It turns out this works very well in BASIC. The reason is the way BASIC handles true and false situations.

So far, every place where you've used a true or false condition has been on an IF statement or a loop of some kind, and the true or false condition occurred because you compared two values. BASIC actually returns a number for a test like this, though. Try

```
PRINT 2 < 1
```

and you'll see that the program prints 1. If you try

```
PRINT 2 > 1
```

the program will print 0. Following this idea, if you try

```
IF 0 THEN
  PRINT "testing..."
END IF
```

you will see that nothing is printed, while

```
IF 1 THEN
  PRINT "testing..."
END IF
```

does print the string.

In fact, BASIC actually accepts a number anytime a condition is expected. If the number is zero the condition is false, while any other value is treated as true. This lets us keep track of true and false values with a normal numeric variable, generally an integer. You can see this idea used in the sample program to keep track of whether or not we have swapped a value; the sample program does this with the variable NOSWAP.

```
REM This program reads in an array of up to 100 real numbers.
REM It then sorts the array, and prints the numbers in order.
REM Numbers are read until a zero is found.
```

```
DIM NUMBERS(99) AS SINGLE :! array to sort
DIM NUM AS INTEGER :! # of numbers actually read
```

```
! read the list of numbers
CALL READEM
```



```
! sort the numbers
CALL SORT
```

```
! write the list of numbers
CALL WRITEEM
END
```

```
!-----
!  
! ReadEm - Read the list of numbers  
!  
! Shared Variables:  
!   numbers - array of numbers read  
!   num - number of numbers read  
!  
!-----
```

```
SUB READEM
```

```
SHARED NUMBERS(), NUM
```

```
DIM RVAL AS SINGLE :! number read from the keyboard
```

```
NUM = 0
DO
  INPUT RVAL
  IF RVAL <> 0.0 THEN
    NUMBERS(NUM) = RVAL
    NUM = NUM + 1
  END IF
LOOP UNTIL RVAL = 0.0
END SUB
```

```
!-----
!  
! Sort - Sort the list of numbers  
!  
! Shared Variables:  
!   numbers - array of numbers read  
!   num - number of numbers read  
!  
!-----
```

```

SUB SORT

SHARED NUMBERS(), NUM

DIM TEMP AS SINGLE :! temp variable; used for swapping
DIM DIDSWAP AS INTEGER :! has a swap occurred?
DIM I AS INTEGER :! loop variable

! loop until the array is sorted
IF NUM > 1 THEN
  DO
    ! no swaps, yet
    DIDSWAP = 0

    ! check each element but the last
    FOR I = 0 TO NUM - 2
      ! if a swap is needed then...
      IF NUMBERS(I + 1) < NUMBERS(I) THEN
        ! note that there was a swap
        DIDSWAP = 1

        ! swap the entries
        TEMP = NUMBERS(I)
        NUMBERS(I) = NUMBERS(I + 1)
        NUMBERS(I + 1) = TEMP
      END IF
    NEXT
  LOOP WHILE DIDSWAP
END IF
END SUB

```

```

!-----
!
! WriteEm - Write the list of numbers
!
! Shared Variables:
!   numbers - array of numbers read
!   num - number of numbers read
!
!-----

```

```

SUB WRITEEM

```

```
    SHARED NUMBERS( ), NUM

    DIM I AS INTEGER :! loop variable

    FOR I = 0 TO NUM - 1
        PRINT NUMBERS(I)
    NEXT
END SUB
```

Try the program a few times to see if you can make it fail. Start with a list of five numbers that are the same. Try a list of five numbers that are already sorted. You might also try the values from the sorting example we worked at the start of this section; the values will be handled internally as real numbers, but INPUT can read an integer and convert it to a real number.

Problem 6.2. The sample program from this section sorts an array so that the smallest number comes first. Sometimes we want the largest number first. Change the sample to it sorts the values with the largest first, proceeding to the smallest.

Problem 6.3. Modify the sample program from the last chapter that reversed the order of characters in a word. This time, sort the characters.

Sort the characters by breaking the string up into individual characters which are stored in an array of strings. Sort the array of strings just like the numbers were sorted, then combine the characters to form the final result string.

You will need to set an upper limit on the size of the string you can sort. Use 255 characters, which also happens to be the largest number of characters the LINE INPUT statement can read from a single line typed from the keyboard.

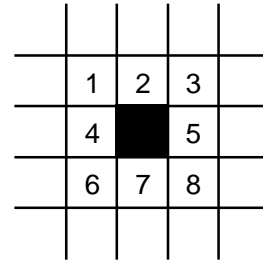
Multidimensional Arrays

The arrays we've dealt with so far are a series of similar values. It's possible to use more than one subscript, though, forming blocks of numbers. There are all sorts of examples of multidimensional arrays from mathematics, especially linear algebra, and from engineering. There's a great example that doesn't use any math at all, though: Conway's game of Life. We'll use that game as a way to introduce multidimensional arrays.

Life is really more of a simulation than a game. It starts with a world consisting of a two dimensional grid with cells, like a sheet of graph paper. Looking at a small chunk of a sheet of graph paper you can see that each cell has eight neighboring cells.

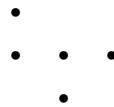
In theory, the number of cells is infinite, extending off in all four directions forever. Life proceeds in generations, filling or emptying each cell based on a simple set of rules.

1. If a cell is filled and has two or three neighbors that are also filled, it stays filled on the next generation.
2. If a cell is empty and has exactly three filled neighbors, it is filled on the next generation.
3. Any other cell will be empty on the next generation.

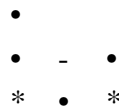


It sounds simple, doesn't it? That's the point. Life was invented to explore how complex systems could become when they are based on a very small number of very simple rules. The results of exploring these simple rules literally fill volumes of information!

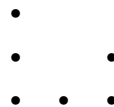
Let's try a seemingly simple example, the r-pentominoe. It's a fancy name for this shape:



Here's the figure as we're getting ready for the second generation. All of the cells that started empty and become filled are marked with an asterisk. All cells that start filled and become empty are shown with a dash. All of the cells that start filled and stay that way are shown with a dot, and of course, all of the cells that start empty and stay that way have no symbol.



Getting rid of the special characters, the second generation looks like this:



same two index numbers, but the order is very important! For our purposes, we can think of the numbers as row and column numbers, with the first index as a column number and the second as a row number. Thinking of the array that way, we could draw it on a piece of paper like you see here.

Each of the squares can be filled in with a distinct value. We refer to a particular value by reading its column number from above the value and its row number to the left of the value. The location for GRID(7, 3) is marked with a spot.

For our Life simulation, though, we'll let each cell represent a distinct cell on the grid. A value of 1 represents a filled cell, while zero represents an empty cell. Here's a program that displays the first fifty generations of the r-pentominoe.

```
REM Conway's game of Life.
REM
REM This version is played on a small, 20 by 20 grid mapped as
REM spots on the Apple IIGS graphics screen.

DIM GRID(21, 21) AS INTEGER :! The current state of the world.
DIM R AS INTEGER , C AS INTEGER :! Row and column number.
DIM G AS INTEGER :! Generation number.

! Set up the world.
FOR R = 0 TO 21
  FOR C = 0 TO 21
    GRID(C, R) = 0
  NEXT
NEXT
GRID(10, 10) = 1
GRID(10, 11) = 1
GRID(11, 11) = 1
GRID(12, 11) = 1
GRID(11, 12) = 1

! Set up the graphics screen.
CALL INITGRAPHICS

! Draw the screen.
CALL DRAWSCREEN
```

```
! Create and move through the generations.
FOR G = 1 TO 50
  CALL NEXTGENERATION
  CALL DRAWSCREEN
NEXT
INPUT " ";A$
END
```

```
!-----
!  
! DrawScreen - Draw the cells on the graphics screen  
!  
! Shared variables:  
!   Grid - array containing the state of the spots  
!  
!-----
```

```
SUB DRAWSCREEN
```

```
  SHARED GRID()
```

```
  DIM R AS INTEGER , C AS INTEGER :! Row and column number.
```

```
  SETPENSIZE (3, 3)
  FOR R = 1 TO 20
    FOR C = 1 TO 20
      IF GRID(C, R) = 0 THEN
        SETSOLIDPENPAT (0)
      ELSE
        SETSOLIDPENPAT (15)
      END IF
      MOVETO (C * 4, R * 4)
      LINETO (C * 4, R * 4)
    NEXT
  NEXT
END SUB
```

```
!-----  
!  
! InitGraphics - Set up for graphics  
!  
!-----
```

```
SUB INITGRAPHICS  
HGR  
SETPENMODE (0)  
SETSOLIDPENPAT (15)  
END SUB
```

```
!-----  
!  
! NextGeneration - Calculate the next generation  
!  
! Shared variables:  
!   Grid - array containing the current state of the  
!         spots; this is updated.  
!  
!-----
```

```
SUB NEXTGENERATION
```

```
  SHARED GRID()
```

```
  DIM WORK(20, 20) AS INTEGER :! Used to generate the next state  
of the world.
```

```
  DIM R AS INTEGER , C AS INTEGER :! Row and column number.
```

```
  DIM N AS INTEGER :! Number of occupied surrounding cells.
```

```
  FOR R = 1 TO 20  
    FOR C = 1 TO 20  
      N = GRID(C - 1, R - 1) + GRID(C, R - 1) + GRID(C + 1, R -  
1) + GRID(C - 1, R) + GRID(C + 1, R) + GRID(C - 1, R + 1) +  
GRID(C, R + 1) + GRID(C + 1, R + 1)  
      IF N = 3 THEN  
        WORK(C, R) = 1  
      ELSE IF N = 2 THEN  
        WORK(C, R) = GRID(C, R)  
      ELSE  
        WORK(C, R) = 0  
      END IF  
    NEXT C  
  NEXT R
```



```
    NEXT
NEXT
FOR R = 1 TO 20
  FOR C = 1 TO 20
    GRID(C, R) = WORK(C, R)
  NEXT
NEXT
END SUB
```

There are a couple of interesting points about the way the program is written. First, notice that we used a 22 by 22 grid, not a 20 by 20 grid. By adding an extra row of cells around the entire grid we were able to simplify the entire program enormously. A single loop handles all of the calculations, even for the corners and edges. Without that extra row we would need an extra chunk of program to deal with the top row, another to deal with the bottom row, a third to deal with the left edge, and a fourth for the right edge. Not only that, but we would need extra code for each of the four corners! That would be nine chunks of code to do the calculations instead of one, making the program longer, harder to write, harder to read, and increasing the chance of making a mistake. One extra row around the outside of the grid is well worth the extra space!

Another point is the way the rules are applied. They aren't quite the same rules we listed earlier in this section. The rules coded in the program do the same thing as the ones listed earlier, but they have been reorganized to fit the way programs are written, not the way people think. That's another trick that often makes a program smaller and easier to write.

When you run the program you'll notice that it's pretty slow. There are many reasons for this. The bottom line, of course, is that the program is doing a lot of work to loop over 400 cells, examining as many as 3600 cells in the process. Still, it seems slow, even allowing for all that work. Part of the reason is that the program itself can be written to run faster using lots of programming tricks. Those tricks apply strictly to this case, though, and not to a broad class of programs you are likely to write, so we won't spend time going over them. Another reason the program is slow is that GSoft BASIC is an interpreter. Interpreters are not as fast as compilers, which in turn are not as fast as hand-coded assembly language. This particular program is a prime candidate for an assembly language subroutine implemented as a user tool. Even versions written with compilers like C and Pascal are slow on a stock Apple IIGS!

Every programmer eventually declares an array that is way too big for the available memory. This sort of problem sneaks up on you, because the numbers involved can look very manageable. For example, you might be tempted to try a three dimensional version of Life, setting up a moderate size grid for experimenting like this:

```
DIM GRID(100, 100, 100) AS INTEGER
```

While this declaration looks innocent, though, it eats memory at a ferocious rate. There are $100*100*100$ numbers, for a total of 1,000,000 values. Integers are one of the smallest number formats you can use, but each does use two bytes of memory. The entire array would take almost two megabytes. (A megabyte is $1024*1024$ bytes, or 1,048,576 bytes.) Two copies of the array would use all of the available memory on a fairly well-equipped Apple IIGS computer, and four copies would burn up all of the memory you can put on the best equipped Apple IIGS!

There is another problem, too. While GSoft BASIC lets you use all of available memory, there is a limit on the size of each individual array. No single array can use more than 32767 bytes of memory. Even if you use several smaller arrays, you can't use more than 65536 bytes of memory for all of your variables unless you use the SETMEM command to expand the available memory.

We won't deal with such large chunks of memory in this course. If you would like more information about how memory is used and how to use the SETMEM command to extend the amount of memory available to GSoft BASIC, see the GSoft BASIC reference manual.

Problem 6.4. Another interesting shape for Life is called a glider. It's shown below. Change the Life program so it follows a glider for ten generations instead of the pentominoe for fifty generations.

```
  •
 •
• • •
```

This is one of the key discoveries in the game of life. A glider moves from one place to another, so it can be used to carry information. After all, what's really the difference in principle between an electron flowing through a wire to carry a bit in a computer chip and a glider moving along a grid? As it turns out, not as much as you might think. Building on ideas like this, researchers have demonstrated that the rules for the game of life are rich enough to construct the same logic circuits that are used in modern digital computers!

Problem 6.5. A matrix is an array of numbers, frequently two dimensional. Linear Algebra defines operations on matrices, just like every day arithmetic defines operations

on numbers. One simple matrix operation is matrix addition, where the corresponding cells in two arrays are added to create a third array. For example, adding these two arrays

1	2	3	1	1	1
4	5	6	1	1	1
7	8	9	1	1	1

gives this matrix

2	3	4
5	6	7
8	9	10

Write a program with three arrays, A, B and C. Each array should hold nine SINGLE values, with two subscripts in each array that range from 0 to 2. Fill in the arrays A and B with the values shown above, then add the two matrices. Print the result and make sure it matches the result you see above.

Hint: Don't try to package the matrix addition as a subroutine. There are some subtle features of BASIC involved that we haven't covered yet.

Passing Arrays to a Subroutine

So far we've used shared variables to use arrays from within subroutines. You can also pass arrays as parameters, and in fact doing so will speed up our program a bit. We'll look at why in a moment.

First, though, let's look at the mechanics of passing an array. If you recall, an array and a variable can share the same name. That forced us to use parenthesis after the name of an array in the SHARED statement, and it forces us to do exactly the same thing for a passed parameter. When you pass an array to a subroutine, place parenthesis after the name of the array in both the subroutine or function call and the parameter list on the SUB or FUNCTION statement. Here's the Life program rewritten to use arrays passed as parameters rather than shared variables.

```
REM Conway's game of Life.  
REM  
REM This version is played on a small, 20 by 20 grid mapped as  
REM spots on the Apple IIGS graphics screen.
```

```

! GRID1 and GRID2 hold the state of the world on alternate
generations.
DIM GRID1(21, 21) AS INTEGER , GRID2(21, 21) AS INTEGER

DIM R AS INTEGER , C AS INTEGER :! Row and column number.
DIM G AS INTEGER :! Generation number.

! Set up the world.
FOR R = 0 TO 21
  FOR C = 0 TO 21
    GRID1(C, R) = 0
    GRID2(C, R) = 0
  NEXT
NEXT
GRID1(10, 10) = 1
GRID1(10, 11) = 1
GRID1(11, 11) = 1
GRID1(12, 11) = 1
GRID1(11, 12) = 1

! Set up the graphics screen.
CALL INITGRAPHICS

! Draw the screen.
CALL DRAWSCREEN(GRID1())

! Create and move through the generations.
FOR G = 1 TO 25
  CALL NEXTGENERATION(GRID1(), GRID2())
  CALL DRAWSCREEN(GRID2())
  CALL NEXTGENERATION(GRID2(), GRID1())
  CALL DRAWSCREEN(GRID1())
NEXT
INPUT " ";A$
END

```

```
!-----  
!  
! DrawScreen - Draw the cells on the graphics screen  
!  
! Parameters:  
!   Grid - array containing the state of the spots  
!  
!-----
```

```
SUB DRAWSCREEN(GRID() AS INTEGER )
```

```
DIM R AS INTEGER , C AS INTEGER :! Row and column number.
```

```
SETPENSIZE (3, 3)  
FOR R = 1 TO 20  
  FOR C = 1 TO 20  
    IF GRID(C, R) = 0 THEN  
      SETSOLIDPENPAT (0)  
    ELSE  
      SETSOLIDPENPAT (15)  
    END IF  
    MOVETO (C * 4, R * 4)  
    LINETO (C * 4, R * 4)  
  NEXT  
NEXT  
END SUB
```

```
!-----  
!  
! InitGraphics - Set up for graphics  
!  
!-----
```

```
SUB INITGRAPHICS  
HGR  
SETPENMODE (0)  
SETSOLIDPENPAT (15)  
END SUB
```

```

!-----
!
! NextGeneration - Calculate the next generation
!
! Parameters:
!   Grid1 - array containing the current state of the spots
!   Grid2 - array containing the new state of the spots
!
!-----

SUB NEXTGENERATION(GRID1() AS INTEGER , GRID2() AS INTEGER )

DIM R AS INTEGER , C AS INTEGER :! Row and column number.
DIM N AS INTEGER :! Number of occupied surrounding cells.

FOR R = 1 TO 20
  FOR C = 1 TO 20
    N = GRID1(C - 1, R - 1) + GRID1(C, R - 1) + GRID1(C + 1, R
- 1) + GRID1(C - 1, R) + GRID1(C + 1, R) + GRID1(C - 1, R + 1) +
GRID1(C, R + 1) + GRID1(C + 1, R + 1)
    IF N = 3 THEN
      GRID2(C, R) = 1
    ELSE IF N = 2 THEN
      GRID2(C, R) = GRID1(C, R)
    ELSE
      GRID2(C, R) = 0
    END IF
  NEXT
NEXT
END SUB

```

As you can see, the only difference between sharing an array and passing it as a parameter is that you need to include the type of the array when you declare the parameter.

Parameters can be passed by value or by reference, as you learned in the last lesson. Arrays are always passed by reference, since you can't use the array in an expression. You can use an element of an array, of course, and we've done that in many of our programs, but you can't add 1 to an array as a whole entity, nor can you do any other operation on an entire array. This is a very important point. It means that every array in BASIC can be changed by any subroutine you pass the array to—a fact we use in the Life program, since the NEXTGENERATION subroutine fills in the GRID2 parameter with the appropriate values for the next generation in the game.

You might be tempted to make copies of array parameters in subroutines and functions, and there are certainly situations where that makes sense. Unlike the case with variables, though, making a copy of an array has a serious downside. When you make a copy of a variable in a subroutine the copy doesn't use much memory. The exact amount depends on the kind of variable and the name, but it's generally about a dozen bytes. If you make a copy of the GRID1 array, though, the values in the array use 22x22x2 bytes of memory, or 968 bytes total. It takes time to set up that array, and the array itself eats up a significant chunk of memory. Copying the array from the one passed as a parameter to the local variable also takes time. It's not something to do lightly!

You might think the impact of copying an array would not be a big deal. You'd be wrong. Take a close look at this new version of Life. In the original version of the program NEXTGENERATION calculates the new grid values, then copies them back into the original grid. In this version NEXTGENERATION doesn't copy the values from GRID2 to GRID1; instead, the main program draws the values directly from GRID2, then calls NEXTGENERATION again, this time passing GRID2 as the current generation. NEXTGENERATION creates the new grid in GRID1, which the main program draws, completing a two-generation cycle. That's why the main loop goes from 1 to 25 rather than 1 to 50, but still draws the same number of generations.

That simple change speeds the program up by about 12%. That may not seem like much, but over the space of an hour you would save seven minutes—and seven minutes is a long time to wait!

Problem 6.6. Redo problem 6.5, this time using a subroutine named ADD to add two matrices.

Lesson Seven – Types and Constants

So far we've concentrated on the mechanics of BASIC programs. We've learned how programs execute and how to use loops and subroutine calls to change the normal flow of a program. Along the way you've become used to three kinds of variables, INTEGER, SINGLE and STRING. In this lesson we'll discuss the other built in types in more detail, learn how to declare types of our own, and learn about a powerful new kind of variable, the record. We'll also learn about constants, which offer a shortcut for an idea we've already used occasionally in the course.

Simple Types and Named Types

The Six Built-in Types

Back in Lesson 2 you got a very brief introduction to the built-in types in BASIC when you learned that you could use the DIM statement to create integer variables using the type INTEGER, single-precision floating point values using SINGLE, and strings using STRING. There are three other predefined types in GSoft BASIC. All six types are shown in the table.

name	character	size	minimum	maximum
BYTE	~	1	0	255
INTEGER	%	2	-32768	32767
LONG	&	4	-2147483648	2147483647
SINGLE	!	4	1.2E-38	3.4E38
DOUBLE	#	8	2.3E-308	1.7E308
STRING	\$	1 to 32768		

There is also a strange seventh type called UNIV. It is only used as the type of a parameter for some Apple IIGS tools. Unlike the other variable types, you can pass any four-byte value at all as a value to a UNIV tool parameter.

The first three types are all different kinds of integer values. As you know by now, this means the value can be a whole number, like 43 or -2, but not a value that lies between whole numbers, like 3.14159 or 2.56. Of the three, INTEGER appears in virtually every implementation of BASIC ever written, LONG appears in most implementations of BASIC that are not restricted in size because they are on a small

machine, and BYTE is rather rare. It appears in GSoft BASIC to support certain values used by the Apple IIGS toolbox.

SINGLE and DOUBLE are the types for floating-point numbers. SINGLE is almost always implemented in BASIC, although the exact range of values varies. Applesoft BASIC, for example, uses 5 bytes for each SINGLE value. Most implementations of BASIC that are not implemented on small computers with limited memory also support DOUBLE, which works just like SINGLE but gives a larger range for exponents and more digits of precision. In GSoft BASIC, SINGLE numbers offer seven digits of precision, while DOUBLE numbers offer fifteen digits.

The whole concept of precision may seem a little strange at first. To get an idea what it means think about paying for something with money. The smallest denomination of money used in the United States is a cent, which is 0.01 dollars. We can't express money with more precision than this using actual currency, so values involving a portion of a cent are rounded or truncated. For example, one third of a dollar would be 33 cents, even though we know there is another one third of a cent not accounted for. Floating-point numbers have the same sort of problem, but the precision is limited to a specific number of digits, not a specific value like 0.01. With the seven digits of precision offered by SINGLE numbers you can represent dollar and cent amounts up to \$99,999.99, for example; or you can represent the mathematical value π to six decimals, 3.141593.

Floating-point numbers also lose overall accuracy as calculations pile on top of each other. Going back to the one-third of a dollar example, if you pay one-third of a dollar three times, you would expect to pay a total of 100 cents. If you actually spend one-third of a dollar three times, though, you will have one cent left. Exactly the same kind of error can pile up as you do calculation after calculation using SINGLE or DOUBLE numbers. Eventually, you may literally see values like 0.9999999 when you know that in theory the value should be 1. There is an entire field of study called numerical analysis that deals with this sort of issue and others related to calculating values on digital computers. We won't go into this field any further, but if the sort of programs you write need accurate calculations with floating-point numbers you can certainly find a lot of reading material!

Finally, if you look closely at the table, you'll notice that the smallest number you can represent with a SINGLE or DOUBLE number is listed as a very small positive number. You can have negative SINGLE and DOUBLE values, of course. The table is showing you how close to zero the number can get. Numbers between the value shown and zero are truncated to zero. The reason this happens is rather complicated; it has to do with the way the numbers are actually stored internally. In a few kinds of programs, though, it's important to know that a number will drop to zero if it gets too small, so you need to know this can happen as you plan your programs.

The last built-in type is the STRING, which you've already learned about.

Problem 7.1. Double-precision floating-point numbers require twice the memory of single-precision floating-point numbers, but there is another difference that is sometimes just as important: Calculations with double-precision numbers take more time.

Write a program that stores 1.2 into one floating-point number and 2.3 into another. Loop over a line that multiplies these and saves the result in a third variable. Use a FOR loop with a LONG control variable so you can loop 100,000 times, which gives a result long enough to time with a watch that displays seconds. Run the program two times, once using SINGLE variables and once using DOUBLE variables, comparing the times.

Some of the programs in this course may seem rather slow, and you might be tempted to think that compilers are the only way to get adequate speed. For some kinds of programs that's true, although many programs run fast enough with an interpreted language. An interesting point, though, is that some programs actually run faster using GSoft BASIC than they do in ORCA/C or ORCA/Pascal! That's because most languages on the Apple IIGS use Apple's floating-point package, SANE, to do calculations. SANE does all calculations using 92 bit numbers, even if you only need the precision of SINGLE calculations. GSoft BASIC has it's own floating-point routines which run much faster than SANE because they only do calculations to the required accuracy.

The TYPE Statement

You can also define your own named types. We'll find many uses for this as the course goes on, but we already have one good one. We've been using true and false values throughout the course in IF statements and loops, and in one case we needed to store a true or false value in a variable. Rather than continuing to piggyback on the INTEGER type we can declare an entirely new one called BOOLEAN, which is the name used for this type of value in many languages, including Pascal. The declaration looks like this:

```
TYPE BOOLEAN AS INTEGER
```

After this statement there is a new type with the name of BOOLEAN that can be used in DIM statements, parameters and function return values, just like we normally use the six predefined types. In this case we've really just defined a new name for INTEGER; any BOOLEAN variable works just like any INTEGER variable. The new name works like a comment, though, reminding us what values we expect to store in the variable.

CONST

In a few of our programs we've used specific values that we might want to adjust at some later date. A good example is the Life program in the last lesson, where we used 20

for the size of the grid and 50 for the number of generations. It would be reasonable to put these values at the top of the program where we could change them quickly. We could do that easily like this:

```
DIM SIZE AS INTEGER  
SIZE = 20
```

There's another way to do it that combines these two statements into a single line.

```
CONST SIZE = 20
```

A `CONST` statement has another advantage over a variable besides just saving a line, though. You can't accidentally change the value of a `CONST` variable anywhere in the program.

The added organization this offers isn't very important in a 50 or even a 500 line program, but many programs are thousands of lines long, and in programs that large, any trick to make the program more organized is worth the effort. We'll start to use `CONST` values in many of our programs for the remainder of the course.

One of them will be for `BOOLEAN` values. Here's two `CONST` statements that fit and in glove with the `BOOLEAN` type from the previous section.

```
CONST TRUE = 1  
CONST FALSE = 0
```

If you try these you'll be in for a surprise, though: You'll get an error message saying you misused a constant. That's because these constants are actually already declared! GSoft BASIC loads the declarations for the Apple IIGS toolbox automatically, and these two constants are declared in the toolbox header file. You can find a list of all of the constants, types and subroutines in the tool interface file by editing the tool interface file itself, which is named `GSoftTools.int`.

Records Store More than One Type

Programs are written to manipulate information of one sort or another. So far I've deliberately kept the kind of information we were using simple, using just a few numbers or a few strings. In many real programs, though, you will mix several kinds of values together to represent a single entity, or it will make more sense to use names rather than array indices to combine values.

Let's look at a classic example, a mailing list. Each entry in a mailing list contains a name, address, city, state and zip code. You might break it down into first name and last name or add more information, but for our example this is enough! If you create a program to handle up to 100 addresses, you would end up with declarations like this:

```
CONST SIZE = 99

DIM NAME(SIZE) AS STRING
DIM STREET(SIZE) AS STRING
DIM CITY(SIZE) AS STRING
DIM STATE(SIZE) AS STRING
DIM ZIP(SIZE) AS LONG
```

Well, this works, but it's cumbersome. Fortunately there is a better way. We can create a new type using a record that contains each of the various pieces of information in a named field, like this:

```
TYPE ADDRESS
  NAMEFIELD AS STRING
  STREET AS STRING
  CITY AS STRING
  STATE AS STRING
  ZIP AS LONG
END TYPE
```

ADDRESS is now a type, just like INTEGER. The various values within the ADDRESS type are called fields; they can be any type at all, including other records. You can use the new ADDRESS type to declare variables or parameters. There is one restriction, though: You cannot return a record from a function. Later we'll learn an easy way around this restriction using pointers.

The name of the first field may seem a little odd. Why not just call it NAME? Looking way back to Lesson 1 you can find the answer: NAME is a reserved word, so we can't use it for a field name, just as we can't use it for a variable or subroutine name.

Returning to the address book, we can declare an array of addresses like this:

```
DIM ADDRESSES(SIZE) AS ADDRESS
```

There are two names involved for each value, the name of the variable and the name of the field within the record. You need to use both names separated by a period. Here's a short section of code that sets up one entry in the ADDRESSES array.

```
ADDRESSES(I).NAMEFIELD = "Byte Works, Inc."  
ADDRESSES(I).STREET = "8000 Wagon Mound Dr. NW"  
ADDRESSES(I).CITY = "Albuquerque"  
ADDRESSES(I).STATE = "New Mexico"  
ADDRESSES(I).ZIP = 87120
```

You can use fields from the record in expressions just like variables. For example, you can print a field like this:

```
PRINT ADDRESSES(43).CITY
```

You can assign one record to another without stepping through each field, as in this set of assignments that might be used in a bubble sort to sort records by zip code.

```
IF ADDRESSES(I).ZIP > ADDRESSES(I + 1).ZIP THEN  
  DIDSWAP = TRUE  
  TEMP = ADDRESSES(I)  
  ADDRESSES(I) = ADDRESSES(I + 1)  
  ADDRESSES(I + 1) = TEMP  
END IF
```

That's the only operation you can perform on an entire record, though. You can't add, subtract, or even compare records; for those kinds of operations you need to work with a specific field. For example, the IF statement shows a comparison of the zip code fields in our address record.

Problem 7.2. Write a program that declares a two variables of type ADDRESS, as shown above. Fill one in with your name and address. After filling it in, copy this record to the second record variable, then print the values from that record.

Lesson Eight – Files

A lot of fun and useful programs never save a file to disk or read from a disk file. Arcade games, some adventure games, many scientific and engineering calculations, and all of the programs you have written so far in this course all read data from the keyboard or do calculations based on internal values. On the other hand, the vast majority of programs do read and write disk files. Spread sheets, word processors, data base programs, many games, GSoft BASIC itself—all of these programs read and write files. This lesson introduces files as used in GSoft BASIC.

An Overview of the Process

Any program that makes use of a file has to go through three distinct steps. They are similar to the steps you go through when you use a program like a word processor, so we'll compare the steps to using a word processor, but don't get too carried away with the analogy—as the text will point out, there are significant differences as well as similarities.

The first step in using a word processor is to either create a new document or open an existing document. That's also the first step in using a file from GSoft BASIC. Whether the file already exists or is a completely new file, though, we always call the process opening a file. You must always open a file before doing anything else to the file.

Once a word processing file is open and before you close the file, you generally edit the file. There are exceptions, of course. Sometimes you open a word processor file to read the file or print the file, and don't make any changes at all. The same is true when you are programming. Once the file is open you either read from the file or write to the file.

The third and final step is to close the file. When you are using a word processor, you close the window that displays the file. Closing the window doesn't mean you exit the word processor, it simply means you are through with the particular document you have closed. There may be other documents open, or you might open another document after closing the first one. Again, files in GSoft BASIC work the same way. The only real difference is how a document is saved. With the word processor, changes are not saved to disk until you issue a save command. In BASIC, information you write to a file is saved as soon as you issue the command that writes that piece of information.

One big difference between a word processor and GSoft BASIC is that GSoft BASIC files are opened for reading or writing, but generally not both. Let's look at how this would work if we were writing a word processor.

When the user opens a document in the word processor, the program would open the document on disk, read the entire document into memory, and close the disk file. From that point on it's the copy in memory the user sees, prints, and changes. Nothing is happening to the disk file at all, and in fact, most word processors will let you eject the disk while the file is open. The disk file isn't needed again until the file is saved, and even then the file might be saved to a new file in a new location. If that happens the original disk file isn't needed again at all.

When the word processor document is saved, the program opens the file. If the updated document is replacing the original copy on disk the next step is a little scary: Everything in the original file is actually deleted! At this point the file the program is about to write to is empty, whether it is a new file in a new location or whether the program is replacing an old file. The word processor writes the entire contents of the file, then closes the file.

Thinking about this process, the word processor actually went through the process of using the file twice. The first time the process was open—read—close, and the second time it was open—write—close. That's typical, although as we will see in this lesson it isn't universal. As it turns out, the way the file is opened is actually quite important. You always open a file for input, output, or both, and it's generally easier to write the program, and the program runs faster, if you don't open the file for both input and output at the same time.

Opening a File for Output

In theory files can be of any length. Basically, that means that there is no fixed limit to the number of things you can put in a file. Of course, there's no free lunch. The information you stuff into a file has to be saved somewhere. In the case of the Apple IIGS it is saved on one of the devices GS/OS handles. This is usually a floppy disk or hard disk, but it can also be a network, a tape drive, a printer, or anything else that GS/OS recognizes.

To write a value to a file you need to open the file for output. You open a file with the OPEN statement, which has this general format:

```
OPEN "myfile" FOR OUTPUT AS #1
```

The file name is a string. In the samples in this course you'll end up using a string constant, like "myfile" in the example, but you can use a string variable as well.

The number at the end is called the file number. You can use any integer value from 1 to 32767 as a file number, and just as with the file name, you can also use a variable. It's this number that identifies the file you have opened; you will use this number in every

command that refers to this file from the time you open it until the time you are finished with the file and close it.

You can open more than one file at a time, so long as the file number you use for each file is unique. Other than the obvious limit of 32767 files imposed by the number of available file numbers, BSAIC doesn't limit the number of files you can have open at one time. Memory and the limits of the GS/OS disk operating system used by the Apple IIGS will have a bigger impact than the number of available file numbers.

Writing to a File

There are several ways to write to a file, and we'll cover the most important ways as we go along. The simplest way to write to a file, though, is to use `PRINT` or `PRINT USING`, writing more or less the same way you would write to the computer screen. The only difference is the addition of the file number, which tells the program both that you are writing to a file instead of to the screen, and which file you are writing to. For example, the command

```
PRINT #1, "Hello, world."
```

writes the string to file number 1.

All of the formatting you are used to works with files just as well as it works on the computer screen.

Closing a File

Once you finish writing the file you need to close it. This is an important step. In many cases a computer language buffers the output, saving the information you think is being written to disk in memory until a significant amount of information builds up, then writing it all at once. The GS/OS disk operating system does the same thing. This makes file output enormously faster than it would be without the buffering, but it also means that some of the information that you think has been written to the disk file may still be in memory when the program finishes. Among other things, closing the file flushes the buffer, writing any buffered information to the disk file.

GSoft BASIC does use file buffering, although you won't find that fact spelled out in so many words in the documentation. Like many aspects of programming, the reference manual doesn't explicitly state facts about internal details that don't matter when you are writing programs. You are supposed to close the file, and if you don't it's a bug in your program—and, in the eyes of people who write operating systems and computer languages, it's your problem if you break the rules, not theirs! This may seem like a

heartless attitude, but there is a reason: details like whether files are cached in a buffer and how big the buffer actually is change from one version of a program to another to take into account problems encountered in programs, changes in operating systems, and even changes in how people use computer languages. By not documenting details like these, or clearly stating that these details can change, the people who write languages are giving themselves some freedom to maneuver when they need to make changes.

Closing a file is pretty simple. You use the CLOSE command with the file number for the file you want to close.

```
CLOSE #1
```

Writing Our First File

Let's put all of this together to create a program that writes ten numbers to a file.

```
REM Write the numbers 1 to 10 to a file on ten separate lines.

! Open the file.
OPEN "temp" FOR OUTPUT AS #1

! Write the numbers.
FOR I% = 1 TO 10
  PRINT #1, I%
NEXT

! Close the file.
CLOSE #1
```

This program creates a new file named temp on your disk drive and writes ten lines to the file. Each line has a number. The file itself is a text file. You can open the file using any program that can read text files, including the editor you use with GSoft BASIC.

Reading from a File

Reading a file is just as easy. To read from a file, you first have to open it for input. The OPEN statement is almost identical to the one you used to write a file, but instead of opening the file for OUTPUT, you open it for INPUT.

Here's a program that opens a text file named temp, reads the first ten lines, and prints them to the screen. The file named temp must exist; if it doesn't, you'll get an error when you run the program.

```
REM Read ten lines from the file TEMP and print the lines.

! Open the file.
OPEN "temp" FOR INPUT AS #1

! Write ten lines from the file.
FOR I% = 1 TO 10
  LINE INPUT #1, LINE$
  PRINT LINE$
NEXT

! Close the file.
CLOSE #1
```

You can open a file for input or output, do some operation on the file, close it, and then reopen it to do some other operation, as this program shows. It opens the file temp and adds a new number to the file before rewriting the file. The only restriction is that the file can't be opened twice unless you close it first. You can open several files at the same time, but they must all be distinct files.

```
REM Read ten lines from the file TEMP, add an eleventh line,
REM and write the results back to the file.

DIM NUMBERS(9) AS INTEGER

! Open the file for input.
OPEN "temp" FOR INPUT AS #1

! Read the lines from the file.
FOR I% = 1 TO 10
  LINE INPUT #1, NUMBERS(I% - 1)
NEXT

! Close the file.
CLOSE #1

! Open the file for output.
OPEN "temp" FOR OUTPUT AS #1
```

```
! Write the old lines to the file.
FOR I% = 1 TO 10
  PRINT #1, NUMBERS(I% - 1)
NEXT

! Write a new line to the file.
PRINT #1, 11

! Close the file.
CLOSE #1
```

Problem 8.1. We used numbers in the example, but the file contains ASCII characters. You can see this for yourself by writing a program that writes strings to the file instead of numbers. Writing the names of the months in the year to a file named temp. Next, open the file for input, read the strings from the file, and print them.

File Names, Directories, Path Names and Folders

File Names, GS/OS and ProDOS

So far we've used the less than descriptive name temp for all of our files. As you start writing larger programs you'll want to use more descriptive names, so it helps to know what the rules are.

GSoft BASIC runs under the GS/OS disk operating system, and supports all file names that the disk operating system itself supports. As far as GSoft BASIC is concerned, a file name is just a string it sends to the operating system. If GS/OS likes the name, GSoft BASIC is happy, too.

GS/OS is actually rather flexible about its rules for file names. It grew out of an older disk operating system called ProDOS, and still supports ProDOS format disks. ProDOS can still read and write GS/OS disks, too, although ProDOS can't properly handle files with a resource fork. Because of this heritage, GS/OS still uses the file name rules for the ProDOS operating system whenever it reads from or writes to ProDOS disks. The rules for ProDOS file names are:

1. A file name starts with an alphabetic character.
2. The remainder of the file name is made up of alphabetic characters, numeric digits, and periods.
3. A file name must have at least one character, and no more than 15 characters.

4. GS/OS does not distinguish between uppercase characters and lowercase characters. In other words, the file names MYFILE, MyFile and myfile all refer to the same file on disk.

GS/OS supports other disk formats, too, like the HFS format disks that are used by the Macintosh operating system. If you are reading or writing files on a disk that is not formatted for ProDOS you can use the file name rules that apply for that kind of disk. For HFS, for example, the rules are rather broad: a file name consists of 1 to 31 characters, and you can't use a colon. We'll stick to ProDOS format names in this course, both because it's the dominant file system on the Apple IIGS computer and because all ProDOS names are also valid on HFS disks, and ProDOS and HFS are the only two file systems that are widely used on the Apple IIGS.

Path Names

The remainder of this section deals with how folders are named and how path names are used to specify particular files on the computer. If you're used to the name directory instead of folder, just keep in mind that they are just two different names for the same thing.

I will assume that you are already familiar enough with your computer to move around using a desktop program like the Finder. In the Finder, the first thing you see on the desktop is a list of the disks, lined up along the right-hand side of the screen. Below each disk is a name. To give the name of a disk in a BASIC program you use exactly the same name, but you start it off with a colon character. For example, the disk where the GSoft BASIC program is located is called GSoft. In a file name, you would type

```
:GSoft
```

Double-click on the disk icon and the Finder will open a window showing the various files and folders. For example, one of the folders is called Samples. If you want to look at a file in the samples folder, you add the name of the folder to the disk name, separating the two with another colon, like this:

```
:GSoft:Samples
```

If the folder contains other folders, you can repeat this process, adding the new folder name to the name you have already accumulated.

Eventually, you will get to the right folder, and you will see the file you want to read. Let's assume that you want to read the file Float from the Benchmarks folder, which is

itself in the Samples folder. Once again, you tack the file name onto the names you already have, using a colon to separate the file name from the name of the disk and folder.

```
:GSoft:Samples: Benchmarks: Float
```

The result is called a full path name. It specifies exactly what file you want to read or write.

Partial Path Names and the Default Prefix

So far our programs just gave a file name. Of course, the computer still writes to a specific place on the disk. When you leave off the name of the disk and any folders, the file name is added to a default directory called prefix 8. Prefix 8 is also called the default prefix. In a desktop program you set the default prefix by using one of the file related commands, like open. When you click on the disk button, it changes the default prefix to the name of a new disk. Opening a folder on the disk adds the name of the folder to the default prefix. Closing a folder, of course, removes the name of the folder from the default prefix. The computer remembers this location, and uses it for all files that only have a file name.

Names in Programs

The process of forming names for the OPEN statement, then, is fairly simple. To get at a file in the default prefix, just use the file name. If the file is in a folder in the default prefix, give the name of the folder, followed by the file name, using a colon to separate the two. If you need to give the name of the disk, too, start off with a colon and the name of the disk, then add the folders and file names, again separated by colons.

If this is new to you, the best thing to do is to practice. The easiest way to practice is with the CAT, EDIT and PREFIX commands, which you can use from the shell. The PREFIX command sets the default prefix. To set the default prefix to :GSoft, for example, you would use

```
prefix :gsoft
```

The CAT command catalogs the current prefix, showing you what files and folders are there. Folders are marked with a file type of DIR in the second column.

You've used the EDIT command all through this course to edit programs, but so far you've used it with file names or to edit the program that is already in memory. The command

```
edit :gsoft:samples:benchmarks:float
```

will load and edit a file using a full path name.

Colons and Slashes

ProDOS uses a slash where GS/OS uses a colon to start path names and separate the names of files, folders and disks. The only difference between the slash and colon is the character used; they do exactly the same thing. One of the design goals of GS/OS was to allow Macintosh format HFS disks to be used directly from an Apple IIGS computer, though, and HFS uses colon for a separator. Worse, HFS allows you to use a slash as part of a file name. Obviously something had to give.

The GS/OS design team decided to switch from the slash character to the colon character, but they actually added the colon without getting rid of the slash. You can still use the slash character as long as you are using ProDOS format disks. Frankly, I use it instead of the colon when I am using an Apple IIGS. I think it is easier to read a path name with slashes, which are easier to pick out of text than colons, and the slash is easier to type because it doesn't involve using the shift key.

But that leaves a lingering problem. If both slashes and colons are allowed, how does GS/OS deal with names on HFS disks that contain slashes? The answer is pretty pragmatic. As GS/OS scans a file name, it looks for the first character that is either a slash or a colon. Once one of these characters is found, that character is used for the separator for the entire path name.

Problem 8.2. Insert your GSoft BASIC disk and use the PREFIX command to set the current prefix to :GSoft:Samples: Benchmarks. Verify that you did it right by using this command to open the Float program:

```
edit float
```

Finding the End of a File

In real programs it's rare to actually know how many values are in a file before you open the file and look. When a program reads a file it uses a function called EOF to find the end of a file. The EOF function takes a file number as a parameter, and returns a Boolean value. The value returned is true if the program has reached the end of the file and there is nothing left to read, and false if there is still information in the file.

You can't open a file for input if the file does not exist. If you try, the program will stop with a run-time error. On the other hand, it is perfectly legal for a file to exist, but not have anything in it. You can create a file with no values by opening it for output but never writing any values to the file. In the case of an empty file, EOF is true right after you open it.

Putting all of these rules together, we will change our sample program from a few sections back to read the temporary file of numbers without knowing in advance how many numbers are in the file. That's a good thing, since there could be ten or eleven, depending on which of the sample programs you ran most recently!

```
REM Read any number of lines from the file TEMP and print the
REM lines.

! Open the file.
OPEN "temp" FOR INPUT AS #1

! Read the lines until we hit the end of file mark.
WHILE NOT EOF (1)
  LINE INPUT #1, LINE$
  PRINT LINE$
WEND

! Close the file.
CLOSE #1
```

Problem 8.3. At this point, you have the tools to merge two files. The basic method is simple: you open one file for input and another for output. You read values from one file, writing them to the other, until you get to the end of the first file. Once the first file is copied you can close it. Next you use OPEN to open the second file, then repeat the process of reading and writing values. The only difference is that you don't open the output file a second time. Once the second file is copied, you close both files.

Write a program that writes the integers 1 to 10 to a file called FILE1.

Write a second program to create a second file, called FILE2, that contains the integers 11 to 20.

Write a third program to read FILE1, writing it to a file called FILE3. It should then read FILE2, adding the contents of FILE2 to FILE3. The program should not depend on knowing the length of either file.

Check your work with yet another program that reads the values in FILE3 and writes them to the text screen.

Problem 8.4. Some folks like uppercase, and some like lowercase. Let's assume that, for some reason, you want to convert the source code for one of your programs to lowercase characters. Change the sample program so it reads a line, converts all of the characters to lowercase, and then writes the line to a new file. Since the new file is a text file, you can open it with a text editor to see if the program worked. Don't try to check the file from GSoft BASIC using the EDIT command, though, since GSoft BASIC will convert most of the program back to uppercase characters in the process of reading the program!

Keep in mind that you can save a program several ways using GSoft BASIC. This program will not work on a program saved with the SAVE command; you must save the program using SSAVE or TSAVE. All of the sample files on the solutions disk are saved with SSAVE, so you can use any of those files as input to your program.

Problem 8.5. One way publishers measure the size of an article or book is by counting the number of words. Of course, they count them by hand, right? Well, you can do it better.

Write a program that asks for the name of a text file. Scan the file, counting the words. For our purposes, a word is defined as any sequence of characters that starts with an uppercase or lowercase character. It includes all of the characters up to the next character that is not an uppercase or lowercase character or a digit. For example, all of the following are words:

```
word stuff v1
```

On the other hand, a number, like 9.6, is not a word.

As an added bonus, keep track of the lengths of the words. Use an array to track how many words of each length appear in the file. Lump any words longer than 30 characters together into a single element in the array, counting them as if the word was actually 30 characters long. That's long enough to handle any word in the English language.

After scanning the file, print the number of words, the number of characters, the average length of a word, and a table showing how many times a word of each length appeared in the file.

Be sure to use long integers for your character counters. After all, an integer can only hold values up to 32767. Each of these lessons has 30,000 to 40,000 characters, not counting the solutions to the problems.

Note: Be careful! You can't divide the character count by the number of words to get the average word length, because the character count includes spaces, commas, periods, and so forth! You must either compute the average from the word length array or keep a separate character counter for characters that appeared in a word.

Test your program by typing the following text into a file and saving it to disk. If you're feeling lazy, this file is on the solutions disk; it's called WordTest.

```
How, now, brown cow.  
single  
i c a b  
thisisaverylongwordtotesttoseeiflongwordsarecaught
```

Leaving out the histogram entries where there were no entries, the results should be:

```
83 characters.  
10 words.  
4 lines.  
The average word length is 5.4.
```

length	number
-----	-----
1	4
3	3
5	1
6	1
30	1

One final note of caution about this problem. In terms of the complexity of the logic involved, this is the hardest problem so far in this course. It's worth spending some time on it to test and develop your skills. If you get tangled up, though, don't hesitate to scrap your program and try another approach. There are relatively easy ways to make this program work, and very hard ways. Don't get stuck struggling with a hard way.

Printing with Files

It is possible to write a driver for almost any input or output device that you can use with GS/OS. One of the most useful examples is one that comes with GSoft BASIC called .PRINTER. As the name implies, this driver is used to send information to printers.

If you have a printer, you have probably used it with two kinds of programs. Desktop programs use a Print... command from the File menu which brings up a dialog filled with printer options. This method of printing is great as far as it goes, but it has severe drawbacks for simple text based programs. AppleWorks classic and Applesoft BASIC are

examples of the other way to use a printer. These programs only send text to the printer, and they generally don't support fonts. The .PRINTER driver works the same way.

You need to install the .PRINTER driver before you can use it. You'll find instructions in the GSoft BASIC reference manual. Once it is installed, using it is a snap. You simply open a file named .PRINTER for output and print. The only real trick is ejecting pages, which you can do by sending the character CHR\$(10) to the printer. Here's an example.

```
OPEN ".PRINTER" FOR OUTPUT AS #1
PRINT #1, "Hello, printer."
PRINT #1, CHR$(10);
CLOSE #1
```

Binary Files

So far all of the files we have written and read have contained text. While text files are common and useful, most of the files you deal with aren't really plain ASCII text files like the ones we've dealt with. In fact, even word processor files put other kinds of information into the file. For example, fonts, sizes, underscores, index entries, tab stops and formatting information are all imbedded in a typical word processor file.

There are several ways to handle all of this information, but the most common is for the programmer to design a way to place the information in the file using bytes, integers or long integers. Since these files contain information that is not simple ASCII text you need some way to distinguish them from text, and some way other than INPUT and PRINT to read and write the files. All of these files are collectively referred to as binary files. There is a binary file type on the Apple IIGS, just like there is a text file type. You'll also find a wide variety of other file types on the Apple IIGS; most of these are binary files, although a few, like the source files used by GSoft BASIC are actually special purpose ASCII text files.

Opening and Closing Binary Files

You open a binary file for BINARY input and output, like this:

```
OPEN "temp" FOR BINARY AS #1
```

Unlike text files opened for INPUT or OUTPUT, binary files are always open for both input and output. If the file already exists, reading will start with the first value in the file, and writing will begin by overwriting the first value in the file. Just as with

opening a text file for OUTPUT, opening a binary file that doesn't exist creates a new, empty file.

The CLOSE command closes a binary file the same way it closes text files.

If you recall, I said opening a file for both input and output was generally harder to deal with and less efficient than opening a file for one or the other. You'll learn some of the intricacies of dealing with reading and writing to the same file as you go through this lesson. The one big efficiency tip for dealing with binary files is to clump read and write operations so that you're not constantly switching between one and the other. The reason has to do with the way files are buffered. When you write a value to the file, it isn't really written right away; instead, the value is stored in an internal buffer. When the buffer is full the entire buffer is written in one chunk. This seems minor, but it can make disk output faster by a factor of 10 to 20! The same thing happens when you read a value. Actually, an entire buffer full of information is read into memory. On your next read, the value is already in memory, so it takes less time to read. If you constantly switch between reads and writes these buffers have to be flushed, either writing a small amount of information to disk or dumping all of the information in the read buffer. That causes the program to slow down considerably.

Writing Binary Files

The first example of file output in this lesson was writing the numbers 1 to 10. Let's return to that example to see how it would work with a binary file.

```
REM Write the numbers 1 to 10 to a binary file.

! Open the file.
OPEN "temp" FOR BINARY AS #1

! Write the numbers.
FOR I% = 1 TO 10
    PUT #1, , I%
NEXT

! Close the file.
CLOSE #1
```

As you can see, the PUT statement looks a lot like a PRINT statement. There are some differences, though.

First, there are two commas. The missing piece of information between the commas is there so you can say where you would like the value to be written in the file. We'll return to that field when we look at random access files later in this lesson; it works the

same way for both binary and random access files. If you leave the value out, as we've done, the PUT statement writes the information to the next available spot in the file. If you're writing a new file, as we are here, the information is appended to the end of all of the information already in the file. That's exactly what happened with the text files from the earlier examples.

The other difference is that PUT will only write one value at a time, and that value must be a variable. The technical term is l-value, which stands for the left-hand value in an assignment statement. You can use anything that you could assign a value too, but you are not allowed to use anything that must be calculated or any constants. Because of this rule these two statements are illegal.

```
PUT #1, , 4 :! illegal
PUT #1, , I% + 1 :! illegal
```

The value that is actually written to disk is written in the same format it is stored in memory. That has two advantages over the text file we wrote earlier. First, a lot of time is saved because the number doesn't have to be converted from its internal binary format to a series of characters and back again. Second, the values are usually smaller, so binary files take less space. A binary representation of an integer value requires the equivalent of two characters worth of space. "Aha," you think. "The character version only took one character of space!" Well, yes and no. First, the text version of our program also had to use at least one more character to separate the numbers. In our example it was an end of line mark. That ties the two methods for the values 0 to 9. Even our simple example had a value of 10, though, so with the end of line mark the text version of the file was one byte longer than the binary version. And you have to admit that only ten values even tie binary files, and in most applications you will use lots of values that need more than one character to represent the value.

There's a third advantage of binary files over text files when floating-point numbers are involved. Unlike integer values, it's very difficult to precisely convert floating point values from their internal binary representation to a text representation and back again and end up with exactly the same value you started with. At the very least it takes a lot of text digits. Binary files don't have that problem. When you write a value to a binary file and read it back into a variable, you're guaranteed to get the same value you originally wrote.

Reading Binary Files

Reading values from a binary file is the mirror image of writing them. You use GET instead of PUT, but other than that everything is the same. Here's an example that reads the file of integers created by the last sample.

```
REM Read numbers from a binary file and print them.

! Open the file.
OPEN "temp" FOR BINARY AS #1

! Read and print the numbers.
WHILE NOT EOF (1)
  GET #1, , I%
  PRINT I%
WEND

! Close the file.
CLOSE #1
```

Reading and Writing Practically Any File

Every file on every modern desktop computer, and almost any other computer, is ultimately made up of a series of eight-bit bytes. In practical terms this means GSoft BASIC can open, read, and write any file you'll find on an Apple IIGS disk. By reading the file as a series of BYTE values, rather than the INTEGER values we used in the examples, you can see the contents of absolutely any file.

Problem 8.6. Write a program that asks for the name of a file, reads the file, and writes it's contents to the screen as a series of BYTE numbers.

Try your program on a text file. How are the ends of lines marked?

More About File Types and File Formats

As you look around at the various files in your computer it's fair to ask what's in them and how you can read and write the files from your programs. Unfortunately, the answer in many cases is that you can't. It's not a limitation in BASIC itself that keeps you from dealing with the files, but rather a lack of information. To read files you have to have a pretty good idea what the format is, and to write them you have to have a very good idea or very good backups! What's the exact format for a WordPerfect word processing file? I have no idea.

There are some places you can go to find information about file types. The single most complete source is *File Type Notes*, a collection of detailed information about the internal contents of dozens of popular file formats, like AppleWorks classic files and several kinds of graphics files. This document was originally created by Apple Computer, and is still available from the Byte Works, Inc. Sometimes you'll find file formats in the

documentation that comes with a program, although that's rare. You can find the formats for public formats like TIFF or JPEG graphics files in various books, from various standards organizations, or by searching the Internet. There really isn't a single repository for all file formats, and there isn't even a guarantee that the file format has ever been documented. Many programmers rely on internal comments within program source code for documentation of file formats.

So why are companies so stingy with information about their files? The reasons vary. In some cases they don't want competitors to be able to read and write their files. You can argue with their logic, but it is a common reason. I think the main reason, though, is simply that it takes a long time to document a file format in a way that someone who is not familiar with the source code for the program can understand, and once the documentation is available, the companies don't really want to have to deal with the inevitable questions from people trying to use the format. Worse still, myriads of programs floating around creating almost correct files, or files that work with one version of the program but end up failing with a later version, could cause the company a lot of grief in terms of customer support.

In any case, there are some kinds of files that you simply won't be able to read or write without spending an enormous amount of time essentially decoding how the file is constructed.

A second issue is the file type itself. Our programs create either text (TXT) or binary (BIN) files. How do you change the file type, or for that matter, how do you tell what type a file is from inside a program? It turns out that detecting and changing file types is so closely tied to the underlying operating system that most languages don't have a way to do it. You have to make calls directly to the operating system, in this case using the GS/OS operating systems GetFileInfo and SetFileInfo calls, to detect or set the file type. That's not something we'll cover in this course, but you'll know enough to strike out on your own after the next lesson. The GS/OS operating system is documented in *Apple IIGS GS/OS Reference*. It was written by Apple Computer and originally published by Addison-Wesley. Reprints are available from the Byte Works, Inc. The whole subject of dealing with GS/OS is also discussed in *Toolbox Programming in GSoft BASIC*. As of this writing it hasn't been published, but is expected out in 1999, again from the Byte Works, Inc.

Random Access

Let's say you have a file with five numbers, 1, 2, 3, 3, and 5. Of course, we want a file with a 4 in the fourth spot. On a short file like this one, we could just read the entire file into an array or linked list, make any changes we want, and write the modified file. If you know you have enough memory to work on the file that way, it's a good choice in any language.

Of course, in real life, we may not have enough memory to handle a file. It isn't uncommon to work with a mailing list with several thousand entries, for example. A reasonable sized record for handling the entries would be about 200 bytes long. A 10,000 person mailing list, then, would take 2,000,000 bytes, which is more free memory than you are likely to find on most Apple IIGS computers.

Let's face it, if you are using a database, you might be willing to wait when you open a file, and wait again when you save the changed file. Asking you to wait while the file is read and written for each change is a bit much, though.

The obvious solution is to open the file for input and output at the same time. You then scan through the file until you find the value that has to be changed, or, if you already know where the value is, jump right to it. You then read the old value, change it, and write the modified value back to the file.

In GSoft BASIC you open a file for random access input and output by opening the file for RANDOM. You still give OPEN a file name and a file number. Unlike opening the file for OUTPUT, though, the old contents of the file are not destroyed if it already exists. There is also one new piece of information. Random access files let you jump right to a specific record within the file, and the only way the file system can do this is if it knows in advance how long each record is. The last piece of information is the length of each record.

Thinking this through, what we're really doing is turning a file into a kind of array. Each entry in the file is called a record, and each of the records is the same size as every other record in the file. If you know the size of each record, you can jump right to a specific record.

While random access file records aren't the same thing as the records you learned to create in the last lesson, in practice it usually makes sense to read and write BASIC record variables from and to the file. After all, if you're writing individual numbers, there really isn't much difference between a random access file and a binary file.

Let's use the mailing list from the last lesson to see how this works. The record we set up looked like this:

```
TYPE ADDRESS
  NAMEFIELD AS STRING
  STREET AS STRING
  CITY AS STRING
  STATE AS STRING
  ZIP AS LONG
END TYPE
```


Opening the file is really the only change between using random access files and binary files, so let's look at what an OPEN statement would look like for a file made up of this kind of record.

```
OPEN "temp" FOR RANDOM AS #1 LEN SIZE
```

There are many ways to choose the value for SIZE. In general you want the smallest value that will hold all of the information you are stuffing into each database record. In virtually all cases, random access files are made up of a series of record variables or numbers that all hold the same kind of information, like a file of ADDRESS records. In that kind of situation, the SIZEOF function is a huge help. SIZEOF takes a single parameter, which can be the name of a type, like ADDRESS, or the name of a variable. Either way, SIZEOF returns the number of bytes used by the variable. Putting it to use in our OPEN statement turns the OPEN statement into this:

```
OPEN "temp" FOR RANDOM AS #1 LEN SIZEOF (ADDRESS)
```

Unfortunately, it isn't quite that simple. The problem is that string values don't occupy a specific amount of space. That's good and bad. We'll see the good points in a moment, but first let's deal with the bad: You can't tell how large the file records need to be without knowing how long the string values will be.

To understand what this means, let's look at exactly how records are stored in a random access file. As long as the record does not contain strings it is simply copied into the file, just like numbers are copied into BINARY files. Strings are actually stored as the location in memory where the string value can be found. When you write a record containing a string to a disk file, this value is converted into an offset past the start of the disk record. If you skip that number of bytes past the start of the record you will find the first character in the string. The string continues until all of the characters have been placed in the file, then a zero byte marks the end of the string. If there is more than one string in the record, the next string starts right after the first, and so on.

This dump of an actual file shows an ADDRESS record. The address shown is

Byte Works, Inc.
8000 Wagon Mound Dr. NW
Albuquerque, NM 87120

```

$000000  14000000 25000000 3D000000 49000000  '   %   =   I   '
$000010  50540100 42797465 20576F72 6B732C20  'PT  Byte Works, '
$000020  496E632E 00383030 30205761 676F6E20  'Inc. 8000 Wagon '
$000030  4D6F756E 64204472 2E204E57 00416C62  'Mound Dr. NW Alb'
$000040  75717565 72717565 004E4D00 00000000  'uquerque NM   '
$000050  00000000 00000000 00000000 00000000  '                   '
$000060  00000000 00000000 00000000 00000000  '                   '
$000070  00000000 00000000 00000000 00000000  '                   '
$000080  00000000 00000000 00000000 00000000  '                   '
$000090  00000000 00000000 00000000 00000000  '                   '
$0000A0  00000000 00000000 00000000 00000000  '                   '
$0000B0  00000000 00000000 00000000 00000000  '                   '
$0000C0  00000000 00000000 00000000 00000000  '                   '

```

The file dump uses hexadecimal values to show the values of the bytes. Hexadecimal numbers use the characters A to F to represent the values 10 to 15; two hexadecimal digits can represent all of the 256 possible values for a byte. The actual values aren't that important, since you can see a text version of the file dump to the right of the hexadecimal values. In the text version it is easy to see how the four string values come after the record itself, which uses the first twenty bytes. Each of the strings shows up in the record as a four-byte hexadecimal offset. The first value is hexadecimal 14, which is the equivalent of the decimal value 20, telling us that the string starts 20 bytes after the start of the record.

If you're head is spinning by now, take heart: The details aren't that important. The important thing you have to remember is that any record that contains strings needs more space in the file than SIZEOF returns as the size of the record. How much more space? You need to add the length of all of the strings as returned by the LEN function, plus one extra byte for each string to store the zero that marks the end of the string.

One way to allow for the extra space is to add the lengths of the longest string that appears in each field. Assuming you've found the length of the largest string for each field and stored the sum in a variable called STRINGLENGTHS, your OPEN statement would look like this:

```

OPEN "temp" FOR RANDOM AS #1 LEN SIZEOF (ADDRESS) +
STRINGLENGTHS + 4

```

You might want to add more entries to your database later, though, and some of those entries may be longer than the ones already in the database. It's a good idea to add some extra bytes to allow for longer fields in the future. Here's the good news about how records are stored: If a record has one exceptionally long string, say a street name, but the other strings are below average length, the strings will still fit in the file records if the

total length is small enough. By using the size of the longest string occupying each field for the size of our records we are building in some extra space, since it is unlikely that any one record will contain the longest string in every one of its fields.

So what happens if the strings are too long? Basically, they are chopped off. Any characters that won't fit in the file record are dropped, and will be missing when you read the record from the file.

Reading and writing random access files works just like it does for binary files. The big difference is that you're more likely to want to read a specific value from the file, so you're more likely to want to use that second parameter for the GET and PUT statement. Here's a GET statement that reads the third record from a random access file, placing the value in a record variable named ADDR.

```
GET #1, 3, ADDR
```

There's a file on the solutions disk called MailingList. We'll use this file for all of the problems that deal with random access files. Here's a program that prints the contents of the mailing list file. It does double duty by showing you how to put all these ideas together into a working program as well as giving you a program that will check your answers for some of the problems. The program itself is also in the solutions disk; it's called PrintList. Both the program and the data file are in a folder called Lesson.8.

```
REM Write the contents of the file MailingList.

TYPE ADDRESS
  NAMEFIELD AS STRING
  STREET AS STRING
  CITY AS STRING
  STATE AS STRING
  ZIP AS LONG
END TYPE

CONST SIZE = 200: ! Number of bytes in one file record

DIM ADDR AS ADDRESS: ! Address read from the file
```

```

! Write all entries in the file.
OPEN "MailingList" FOR RANDOM AS #1 LEN SIZE
WHILE NOT EOF (1)
    GET #1, , ADDR
    PRINT ADDR.NAMEFIELD
    PRINT ADDR.STREET
    PRINT ADDR.CITY;" , " ;ADDR.STATE;" " ;ADDR.ZIP
    PRINT
WEND
CLOSE #1

```

Problem 8.7. Write a program that opens the MailingList file and prints the 3rd record from the file.

Problem 8.8. Write a program that let's you type new values from the keyboard, then stores those values in a record, writing the record at the end of the current MailingList file. (Be sure to make a copy of the file first!)

Problem 8.9. The folks in the marketing department keep running across names that are so long that the current record size is causing problems. Write a program that reads the MailingList file and writes it to a new file whose records are ten bytes longer. Check your work with a modified form of the PrintList program that appears in this section as a sample program.

Lesson Nine – Pointers and Lists

What is a Pointer?

By now, you have used two very powerful techniques to organize information in BASIC. Arrays are used to handle a large amount of information when all of the pieces are the same type. Records are used to collect different kinds of information into a single variable.

While these types are very powerful, there is one situation they do not handle well. In many programs you don't know in advance how many pieces of information you need to deal with. For example, a program to manage a mailing list may have a few hundred entries when one person uses it, but several thousand for another person. One solution is to allocate an array that will be big enough to hold some maximum number and leave it at that. Of course, that presents a problem, too. If one person has a computer with 1.25M of memory, they may be able to handle a mailing list with 7000 or 8000 entries. Unfortunately, the program would be too large to run on a computer with 768K, and would not make effective use of all of the memory in a 2M machine.

Of course, you may not ever intend to write a commercial application. On your own machine, you know how much memory you have, right? Well, that could be true, but fixed size arrays present other problems. Many programs have to handle more than one kind of data at the same time. For example, an adventure game might need one array for handling the rooms in a castle, and another array for keeping track of the various inhabitants. You can try to make effective use of memory by guessing in advance how big each array needs to be, but if you guess wrong, you could overflow one array while there is still plenty of room in the other.

And, of course, all of this ignores the fact that the current implementation of GSoft BASIC on the Apple IIGS limits the maximum size of a single array to 32K.

In all of these situations, the problem is that you know there is a lot of memory out there, but you don't always know, in advance, how much memory is available or exactly what you will need to use it for when the program runs. The amount of memory used by an array or record is determined when the program is written. You can't change it without changing the program itself. What we need is a way to ask for a chunk of memory while the program is running. Programmers call this *dynamically allocated memory*. Since GSoft BASIC doesn't know where the memory will be when you compile the program, or even how much will be allocated, you need some way of keeping track of the memory. That, in a nutshell, is what pointers are for. A pointer *points to* a memory location. In terms of the BASIC program, a pointer points to a variable. The variable can be a simple variable, like an integer or a real number; a record; an array; or even another pointer. In

short, a pointer can point to a variable of absolutely any type except an array—and it can point to an element of an array, or a record containing an array.

I don't want to scare you off, but pointers tend to give beginners a lot of trouble. I would like to talk for a moment about what kind of trouble people have so you can watch out for these issues as you read through the lesson. We will try to deal with each of the issues.

Part of the reason people have trouble with pointers is that the idea of dynamically allocated memory is foreign to those of you who cut your teeth on traditional implementations of BASIC, which don't support pointers. If pointers are a new concept for you, you should expect it to take some time before you become comfortable with them. Another factor is that pointers have their own operator that you must learn to use. A lot of people get confused by this operator, which controls when you are dealing with a pointer, and when you are dealing with the thing it is pointing to. Finally, there is a bit of magic about pointers in a high-level language. The other data types we have dealt with were definite, fixed structures. You could get a handle on what they do, and how they work. From a language like BASIC, there are some mysteries to how pointers work, since the language takes care of a lot of details. It is only from assembly language that you really see how pointers work—and, if you ever learn enough assembly language to learn how pointers work, you will probably follow in the footsteps of the vast majority of programmers, and return to a language like BASIC that handles all of those mucky details for you!

A realistic example of how pointers are used in a real program is well beyond what you are likely to understand at this point, so some of the first few examples will seem very simplistic and contrived. You will look at them and wonder why we are using pointers at all, when you can easily see better ways to write the program without a pointer. Well, you are right, but we will use some simple programs to get used to the mechanics of pointers. By the end of the lesson, though, you will be dealing with data structures that you could not handle with arrays. In the next few lessons, we will start doing things with pointers that are very difficult to do with arrays. In some cases, in BASIC at least, some of the things we will do can't be done any other way than by the use of pointers. That's especially true if you continue on to toolbox programming after this course. The Apple IIGS toolbox is littered with various kinds of pointers.

Pointers are Variables, Too!

The first thing we need to explore is how to define a pointer. Like an array, which must be an array of something, a pointer must point to something specific. You can't define what a pointer points to using a type character on the variable name as we have done with simple variables. You always use a TYPE statement to declare a pointer type and a DIM statement to create a pointer variable.

```
DIM IP AS POINTER TO INTEGER
```

The variable IP is a variable, just like any other. It just has an odd type. The type of IP is `POINTER TO INTEGER`. There are only two things that you normally do with this variable in BASIC: assign it to another pointer variable or compare it for equality or inequality with another pointer value. Of course, for either operation, the pointers must point to the same kind of value. For example, the following program is legal in BASIC:

```
DIM IP AS POINTER TO INTEGER
DIM JP AS POINTER TO INTEGER

IP = JP
```

The pointer is virtually worthless without the `^` operator. The `^` operator, appearing right after the pointer variable, gives us the value the pointer points to rather than the pointer itself. For example, the assignments shown in the following program are legal, although the program itself has some problems. **Do not run this program!**

```
DIM IP AS POINTER TO INTEGER
DIM I AS INTEGER
DIM J AS INTEGER

J = 4
IP^ = J
I = IP^
PRINT I
```

Let's step through the program, looking at what it is doing. First, we assign the value 4 to J. Nothing is new there; you've done that sort of thing dozens of times. The next line, though, assigns the integer J to the value pointed to by IP. Keep in mind that we are not assigning a value to the variable IP, we are assigning a value to the variable *pointed at* by the variable IP. That's what the `^` operator does for us; it tells BASIC that we want the value pointed at, not the pointer.

If that's confusing, think about how records work for a moment. If A and B are the same kind of record, then the assignment

```
A = B
```

copies the contents of the record B into the record A. This is very, very different from the assignment

```
A.LEFT = 4
```

which copies the value 4 into one field of the record. The .LEFT tells the program to use a specific field from the record, not the record itself. The ^ operator is doing something similar for a pointer variable. It tells the program to copy the value into the variable pointed to by IP, not into the pointer IP itself.

The next line,

```
IP^ = J
```

uses the same idea to assign the value pointed to by IP to the variable I. Finally, the value of I is printed. The value should be 4.

Unfortunately, this program has a very, very serious flaw. It is a very common error in programs that use pointers. In fact, it is one of the most common causes of crashes on the Apple IIGS, in any kind of program, in any language. Did you catch the flaw? If you've never seen pointers before, probably not.

What does IP point to?

What if IP points to the location in memory that turns on your floppy disk drive? The disk drive would start to spin.

What if IP happens to point to memory allocated by the GS/OS operating system that holds a block of a data file? When you save the file, it will have some garbage information in it.

What if IP points into the middle of your program? Your program may crash.

Worst of all, what if IP points to some memory that isn't being used for anything? You might think the program works, and pass it around to friends. It could then do all of these nasty things to *their* computer. This, of course, is not a good way to keep friends.

Allocating and Deallocating Memory

In short, pointers are no good without a way to get some memory for them to point to. BASIC gives us a statement called ALLOCATE to get some new memory. When you are finished with the memory, the DISPOSE statement can be used to get rid of the memory. Both statements need the name of the pointer for which you want to allocate or deallocate memory. We can change our program from the last section into a safe one using these procedures. This program is one you can run!


```
DIM IP AS POINTER TO INTEGER
DIM I AS INTEGER
DIM J AS INTEGER

ALLOCATE (IP)
J = 4
IP^ = J
I = IP^
PRINT I
DISPOSE (IP)
```

When this program runs, it starts by making a call to `ALLOCATE`. This statement performs some advanced magic. The result is that, after the call, two bytes of memory have been obtained. The exact process involved in getting this memory is a bit involved, and not particularly important to you, the BASIC programmer. The process is covered below, just in case you're curious. In any case, this memory is safe. It belongs to your program, and no other correctly written program will disturb it.

Just before the program ends you see the `DISPOSE` statement. This statement goes through a complicated mechanism that gets rid of the two bytes of memory. After calling `DISPOSE`, the memory does not belong to your program anymore. It could be reused within 1/60th of a second by an interrupt routine, which is a small program that does things like tracking the mouse or reading the keyboard in the background while your program runs. Even if it isn't reused, because of the process used to allocate and deallocate memory, the location `IP` points to doesn't contain 4 anymore. In short, once you call `DISPOSE`, the memory isn't yours anymore, and you should not access or change the value pointed at by `IP`.

How New and Dispose Work

The process used to allocate and deallocate dynamic memory is a bit involved, and has nothing in particular to do with the way you write your BASIC program, but it is interesting. If it's not interesting to you, though, you can safely skip this entire description.

One of the basic parts of the Apple IIGS operating system is the Memory Manager. The Memory Manager is responsible for finding free memory and giving it to the various programs in the computer. Even if your program is the only one you think is running, it turns out that many other programs are calling the Memory Manager to get memory, too. The GS/OS disk operating system calls the Memory Manager, as do many of the Apple IIGS tools. GSoft BASIC is calling the Memory Manager to get space for your program. Many desk accessories call the Memory Manager. Some of them may even

install interrupt handlers, which can be running while your program is doing something else.

When you call `ALLOCATE` for the first time, GSoft BASIC makes a call to the Memory Manager to get a 4K block of memory. This memory is then subdivided into smaller and smaller pieces, dividing the block in half each time, until the program gets a chunk of memory of about the right size. In our program you need two bytes to hold the integer, and the library subroutine allocating the memory needs four bytes to keep track of all of the small pointers, so a total of eight bytes is actually taken from the 4K chunk of memory. (Remember, the number of bytes will be a power of two.) This method tends to waste a few bytes of memory now and then, but it turns out that it is very fast. It has some other technical advantages, too, that we won't go into here.

When you call `DISPOSE` at the end of the program the small block of memory is deallocated. Since it was the only piece of memory being used in the 4K block, the 4K block is also returned to the Memory Manager, where it can be reused by other programs. If you had allocated other pieces of memory in addition to the one IP points to, and those were still in use, the block would not be deallocated and returned to the Memory Manager until all of the individual pieces were disposed of.

An interesting point about this memory is where it comes from. Unlike variables, arrays, and even strings, memory allocated by calling `ALLOCATE` doesn't come from the fixed size variable space your program allocates when it starts. `ALLOCATE` gets memory directly from the Memory Manager. It will continue to allocate memory until all of the available memory in the Apple IIGS is used.

Problem 9.1. A pointer can point to any variable type. Use that fact to change the program shown in this section to allocate a pointer to a real number. Assign the value 1.2 to the location pointed to by the pointer, and print the result. Do all of this without an intermediate real variable; in other words, assign the value directly to the value pointed at by the pointer, and use the pointer with the `^` operator in the `PRINT` statement.

Problem 9.2. You can, of course, use `IP^` anywhere that you could use an integer variable. Making use of that fact, write a program to add two numbers and print the result. The only variables you should define are three pointers, `IP`, `JP`, and `KP`. Be sure and allocate memory for all of them using `ALLOCATE`, then assign 4 to the first, and 6 to the second. Add the two values together and save them at `KP^`, then print the result. Be sure and follow your mother's advice, and clean up after yourself by calling `DISPOSE` to deallocate the memory areas reserved by the calls to `ALLOCATE`.

Linked Lists

So far all of our programs have used a pointer to a single variable. That's about as useful as your mother on a hot date. A single variable is easier to use, takes less space, produces a smaller program, the resulting program runs faster, and there is no chance of stepping on someone else's memory because you forgot to use `ALLOCATE` to allocate the memory. We used arrays to organize a fixed number of values into a data structure that was easier to use. The equivalent for a pointer is one of the many forms of a linked list.

Basically, a linked list is a series of connected records. Each of the records in the linked list contains, among other things, a pointer. The pointer points to another record in the list. A single pointer variable in the program points to the first record in the linked list.

For our first look at a linked list, we will create a list of integers. The record, then, must have a pointer to the next record, and an integer. It looks like this:

```
TYPE LISTRECORD
  NEXTP AS POINTER TO LISTRECORD
  I AS INTEGER
END TYPE

TYPE LISTPOINTER AS POINTER TO LIST

DIM FIRST AS LISTPOINTER
DIM TEMP AS LISTPOINTER
```

With these definitions we can start to create a linked list. For each element in the list we will need to call `ALLOCATE` to get space for a new record, and then place a value into the integer, like this:

```
ALLOCATE (TEMP)
TEMP^.I = 4
```

Look carefully at the assignment that places a 4 in the record. The characters `^.` may seem confusing at first, but they are the same simple ideas you are used to, combined to do something a bit more complicated. `TEMP`, of course, is a pointer, so to put a value into `TEMP` we need to use the `^` operator. `TEMP^` points to a record. To place a value into the field `I` within a record we add `.I`. The whole expression, `TEMP^.I`, then, refers to the integer variable `I`, located inside a record that is pointed to by the pointer `TEMP`.

That's a complicated concept, but it is simple when you break it down into parts, reading the expression one symbol at a time from left to right, the way BASIC itself does.

At this point we have a dynamically allocated record with an integer value in it. The pointer in the record still does not point to anything. The next step is to add this record to the list of records that the variable `list` points to.

```
TEMP^.NEXTP = FIRST
FIRST = TEMP
```

On the first line we are assigning a value to the pointer in our new record. The value we are assigning is `FIRST`; `FIRST` points to the first element currently in the list. We really don't know how many things are in the list at this point. There may not be any, or there may be several thousand. The beauty of the linked list, though, is that we don't have to know! It doesn't matter at all how many things are already in the linked list.

The second line assigns `TEMP` to `FIRST`. The first thing in the list, at this point, is our new record. Our record contains an integer variable with a value of 4, and a pointer to the rest of the list.

The next thing we need to learn is how to take something off of the list. Let's say that we want to remove the first item. Basically, then, we reverse the process of putting a record into the list, like this:

```
TEMP = FIRST
FIRST = TEMP^.NEXTP
```

There is one more detail that we need to deal with before we can use these ideas to write a program. So far we have ignored the issue of the end of the list. How do we know when we get to the end of the list? We could keep a counter, but actually there is a better way. It involves the use of a predefined pointer constant called `NIL`. `NIL` is type compatible with any pointer type. You can set a pointer to `NIL` or compare a pointer to `NIL`. By convention, `NIL` is used to mean that the pointer doesn't point to anything, and that is how we mark the end of our list. By initializing `list` to `NIL` at the start of the program and checking to see if `list` is `NIL` before removing an item from the list, we can tell when there is nothing in the list.

Stacks

Using what we now know about linked lists, we can create our first program.

```
REM This program reads in a first of integers, and then prints
REM them in reverse order. The program stops when a zero
REM value is read.
```

```
TYPE LISTRECORD
  NEXTP AS POINTER TO LISTRECORD
  I AS INTEGER
END TYPE
```

```
TYPE LISTPOINTER AS POINTER TO LISTRECORD
```

```
DIM FIRST AS LISTPOINTER: ! points to the top item in the first
```

```
CALL GETLIST(FIRST): ! read a list
CALL PRINTLIST(FIRST): ! print a list
END
```

```
!-----
!  
! GetList - Read a list from the keyboard  
!  
! Parameters:  
!   first - pointer to the head of the list  
!  
!-----
```

```
SUB GETLIST(FIRST AS LISTPOINTER)
```

```
  DIM I AS INTEGER : ! value read from the keyboard  
  DIM TEMP AS LISTPOINTER: ! work pointer
```

```
  ! initialize the list pointer  
  FIRST = NIL
```

```
DO  
  ! read a value  
  INPUT "Enter a number: "; I  
  IF I <> 0 THEN  
  
    ! allocate a record  
    ALLOCATE (TEMP)
```

```

        ! place i in the record
        TEMP^.I = I

        ! put the record in the list
        TEMP^.NEXTP = FIRST
        FIRST = TEMP
    END IF
LOOP UNTIL I = 0
END SUB

!-----
!
! PrintList - Print a list
!
! Parameters:
!   first - pointer to the head of the list
!
!-----

SUB PRINTLIST(FIRST AS LISTPOINTER)

DIM TEMP AS LISTPOINTER: ! work pointer

WHILE FIRST <> NIL
    ! remove an item from the list
    TEMP = FIRST
    FIRST = TEMP^.NEXTP

    ! print the value
    PRINT TEMP^.I

    ! free the memory
    DISPOSE (TEMP)
WEND
END SUB

```

We have already talked about all of the ideas in this program, this is just the first time you have seen them all in one place. Looking through the program, the first step is to get a list of numbers. GetList does this, reading numbers using familiar methods until you enter 0. For each number, GetList allocates a new record, saves the number in the record, and puts the record in the list.

PrintList loops for as long as there are entries left in the list. Each time through the loop the top record in the list is removed from the list, the value is printed, and the memory used by the record is dumped.

Notice how the PrintList procedure cleans up after itself. The memory used by every record is carefully disposed of after we are finished with the record. This is an important step in a program that uses dynamic memory. If you forget to dispose of some of the memory in a few places, the memory areas will eventually fill up, and there won't be any free memory for new calls to ALLOCATE. This is known as a memory leak.

It is very important to understand exactly how this program works, since the ideas used in this program form the basis for many of the fundamental techniques in modern programming practice. Stop now, and type in the program. Run the program with the following input:

```
1
2
3
4
0
```

The program responds with this:

```
4
3
2
1
```

This may not have been exactly what you expected. What happened is this: When the program creates the list, each new element is added on top of the old list. As the program retrieves records from the list, the last one added is removed first. This mechanism is called a stack. The common analogy is to think of it like a stack of plates. You pile the list elements up on top of one another. To get one back, you pull the top record off of the stack.

Just as a footnote, I should warn you about terminology buffs. Many high school teachers, a few college professors, and even an occasional book author figure that the way to become a good programmer is to learn a bunch of arcane words. It is true that you need some new words, like dynamically allocated memory, to describe new concepts, but these terminology buffs want you to know that a stack is called a LIFO data structure, for Last In, First Out. Let's face it, they write the tests, so you better know the term if you want to get a good grade in a class. Be warned, though: if you walk up to a group of programmers at a conference and start babbling about LIFO data structures, you will find

a wide gap forming around you. A few people will glance at your shirt pocket, looking for the pencil holder, or examine the thickness of your glasses. In real life, these things are called stacks.

Stacks are a very flexible data structure. They are used in a wide variety of applications. A stack is appropriate any time you need to collect a large amount of information, especially if you don't particularly care in what order you use the information, or for the occasional case when you want to handle the most recent piece of information first. Stacks are also frequently used as a part of a more complicated data structure, like a hash table. We'll look at complex data structures like this later in the course. Stacks are used in such diverse applications as burglar alarms, data bases, mailing lists, operating systems, and arcade games.

There are many variations on the basic ideas covered in this section. Some of these are explored in the problems. I highly recommend that you work both of these problems.

Problem 9.3. Many applications require you to process the information in a list from back to front. In some cases, you know this in advance, and a slightly different form of a list is used, called a queue. That situation is covered in the next section. In other cases, though, you may not know that the list needs to be reversed in advance, or you may need to process the list in both orders in different parts of the program. In a case like that, you need to be able to reverse the list.

Reversing a list is really quite easy. To do it, you use two lists. The new list starts out empty. You then loop through the old list, just like we do in the PrintList procedure, but instead of printing the value and disposing of the record, you add the record to the new list.

Write a procedure to reverse the order of a list. Use this procedure in the sample program so it prints the numbers in the same order they are read.

Problem 9.4. In some applications we read in a list, then scan the list repeatedly, looking for records with certain characteristics. For example, in a burglar alarm, we might use one subroutine to add new alarms to a list. Another might repeatedly scan the list, looking for fires. If no fires were found, the list could be rechecked for broken windows, and so on.

Implement this idea in our sample program by counting the number of times a particular number appears in the list. Use a FOR loop to loop from 1 to 5. For each value, scan the list, incrementing a counter if the number is found. Print a table of the results.

Try this program at least two times. The first time, enter zero immediately. The second time, use this data:

1
2
3
4
5
2
3
4
5
3
4
5
4
5
5

The results should be one one, two twos, and so forth.

Hint: To scan a list, set a pointer to the head of the list. Use a WHILE loop to loop until this pointer is NIL. At the end of the WHILE loop set the pointer to the next record, like this:

```
TEMP = TEMP^.NEXTP
```

Queues

Another commonly used form of a list is the queue. A queue looks just like a stack, but it is formed differently. A queue is used when you want to process information in the same order it is read, so instead of adding new records to the beginning of the list, you want to add them to the end of the list. In a sense, the records are lined up and processed on a first-come, first served basis. The terminology freaks call a queue a FIFO list, for First In, First Out, but again, don't embarrass yourself in a crowd by talking about stuff like that.

There are three basic ways to form a queue. If all of the information is read in first, then processed, you could just use the simple stack to read the data, then reverse the order of the list, like we did in problem 9.3. In many programming situations, though, you read some data, process a little bit, read some more, and so forth. In those cases, you need to build the list in the proper order.

One way to build a queue is to keep a second pointer, which we will call LAST. This pointer starts at NIL, like the pointer that points to the first member of the list. When we

add the first element to the list the pointer LAST is set to the value of the new pointer. The next pointer in the new record is always set to NIL. From then on, we add a new record by setting the next pointer in the record pointed to by LAST to point to the new record, and then set LAST to point to the new record.

In BASIC code, then, we set the list up like this:

```
FIRST = NIL
LAST = NIL
```

To add a record to the end of the list, we check to see if the record is the first one in the list. If so, we set both LAST and FIRST to point to the new record. If not, we chain the record to the end of the list.

```
IF LIST = NIL THEN
  FIRST = TEMP
  LAST = TEMP
ELSE
  LAST^.NEXTP = TEMP
  LAST = TEMP
END IF
```

Of course, since both branches of the IF statement assign TEMP to LAST, we can make the program shorter and still do the same thing by pulling the assignment outside of the IF statement, like this:

```
IF LIST = NIL THEN
  FIRST = TEMP
ELSE
  LAST^.NEXTP = TEMP
END IF
LAST = TEMP
```

We also don't actually make use of LAST before it is assigned a value for the first time, so setting it to NIL when we initialize the list is also unnecessary.

Problem 9.5. You probably saw this one coming. Change the GetList procedure from the sample in the last section so it forms a queue instead of a stack. Use the mechanism described in this section to do it.

Running Out Of Memory

What happens if you ask for more memory, but none is available? If this happens, `ALLOCATE` sets the pointer to `NIL` rather than to a valid memory location. Just for fun, the following program does this on purpose.

After running this program, quit GSoft BASIC and reenter the program. That cleans up the memory the program allocated and never disposed of. Also, be aware that this program could run for a very long time, especially if you have a lot of memory.

```
DIM P AS POINTER TO INTEGER
DIM COUNT AS LONG

COUNT = 0
DO
  ALLOCATE (P)
  COUNT = COUNT + 1
LOOP UNTIL P = NIL
PRINT COUNT;" integers were allocated."
```

The practical ramifications of this program are very important. In real programs you need to make sure a call to `ALLOCATE` really worked. That means you need to check after each and every call to see if `ALLOCATE` returned `NIL`. If it did, your program has to do something to handle the situation. That might mean reporting an error and quitting, disposing of some buffers you no longer need, or informing the user that an operation can't be carried out. The one thing you can't do is ignore the problem!

Lesson Ten – Miscellaneous Useful Stuff

The first nine lessons of this course have taken you on a tour of the BASIC language. By this time you have learned most of the mechanics of the language itself. Because the lessons have been developed using specific examples, though, a few topics have slipped through the cracks. This chapter covers those topics.

I don't want you to get the impression that these topics are unimportant. Quite the contrary: a great deal of the power of the BASIC language is tied up in the topics we will look at in this lesson. In our tour of the BASIC language, though, we have concentrated on the mechanics of writing short, simple programs. As we learn more about writing larger programs, programming efficiently, and organizing programs, the new techniques covered in this lesson will be put to use over and over.

The SELECT CASE Statement

You've learned to use IF and ELSE IF to select from a series of possible conditions. Here's an example that accepts a number from 1 to 13, representing the value from a deck of cards, and prints the name of the card.

```
!-----  
!  
! PrintCard - Print the name of a card  
!  
! Parameters:  
!   V - point value of the card  
!  
!-----  
  
SUB PRINTCARD (V AS INTEGER)
```

```

IF V = 1 THEN
  PRINT "Ace";
ELSE IF V = 2 THEN
  Print "Two";
ELSE IF V = 3 THEN
  Print "Three";
ELSE IF V = 4 THEN
  Print "Four";
ELSE IF V = 5 THEN
  Print "Five";
ELSE IF V = 6 THEN
  Print "Six";
ELSE IF V = 7 THEN
  Print "Seven";
ELSE IF V = 8 THEN
  Print "Eight";
ELSE IF V = 9 THEN
  Print "Nine";
ELSE IF V = 10 THEN
  Print "Ten";
ELSE IF V = 11 THEN
  Print "Jack";
ELSE IF V = 12 THEN
  Print "Queen";
ELSE IF V = 13 THEN
  Print "King";
END IF
END SUB

```

BASIC has a special statement called the SELECT CASE statement that is used in situations like this. The SELECT CASE statement is like a multiple branch. It works the same as the series of IF and ELSE IF checks, but there is a little less typing and the program runs a little faster. Using a SELECT CASE statement the PrintCard subroutine becomes

```

!-----
!
! PrintCard - Print the name of a card
!
! Parameters:
!   V - point value of the card
!
!-----

```

```
SUB PRINTCARD(V AS INTEGER )

SELECT CASE V
  CASE 1
    PRINT "Ace";
  CASE 2
    PRINT "Two";
  CASE 3
    PRINT "Three";
  CASE 4
    PRINT "Four";
  CASE 5
    PRINT "Five";
  CASE 6
    PRINT "Six";
  CASE 7
    PRINT "Seven";
  CASE 8
    PRINT "Eight";
  CASE 9
    PRINT "Nine";
  CASE 10
    PRINT "Ten";
  CASE 11
    PRINT "Jack";
  CASE 12
    PRINT "Queen";
  CASE 13
    PRINT "King";
END SELECT
END SUB
```

When the `SELECT CASE` statement executes, it starts by evaluating the expression that comes after `CASE`. In our example, the expression is a simple one, consisting of a single variable. The next statement executed is the one right after the value that corresponds to the value of the expression. You can put more than one statement there, of course, even though we only used one statement after each `CASE` label in this example. As soon as the next `CASE` label is encountered the program skips to the statement after the `END SELECT` statement. In other words, the `SELECT CASE` statement works exactly like a series of `IF ELSE` clauses. The `SELECT CASE` statement is just a bit easier to read, and gives you another way to organize your program.

The PrintCard subroutine shows the classic way to organize a SELECT CASE statement, but in situations like this one where there is a single value to check and a single thing to do for each specific value, I like to use the : statement separator to combine the CASE statement with the statement that handles the condition, like this:

```
!-----  
!  
! PrintCard - Print the name of a card  
!  
! Parameters:  
!   V - point value of the card  
!  
!-----  
  
SUB PRINTCARD(V AS INTEGER )  
  
SELECT CASE V  
  CASE 1: PRINT "Ace";  
  CASE 2: PRINT "Two";  
  CASE 3: PRINT "Three";  
  CASE 4: PRINT "Four";  
  CASE 5: PRINT "Five";  
  CASE 6: PRINT "Six";  
  CASE 7: PRINT "Seven";  
  CASE 8: PRINT "Eight";  
  CASE 9: PRINT "Nine";  
  CASE 10: PRINT "Ten";  
  CASE 11: PRINT "Jack";  
  CASE 12: PRINT "Queen";  
  CASE 13: PRINT "King";  
END SELECT  
END SUB
```

Personally, I think this makes the program a lot easier to understand.

There are many situations where you will want to use several different case labels for the same statement. To do this, separate the case labels with a comma, as the following example shows.


```
FOR I = 1 TO 10
  SELECT CASE I
    CASE 1, 2, 3, 5, 7
      PRINT I; " is prime"
    CASE 4, 6, 8, 10
      PRINT I; " is even"
    CASE 9
      PRINT I; " is odd"
  END SELECT
NEXT
```

While listing specific values is appropriate for the majority of SELECT CASE statements you're likely to write, there are two ways to handle ranges of values. The first is to give start and end values for a range of values, separated by the word TO. The second is useful for collecting all of the remaining values that have not been picked off by a specific CASE statement. The CASE ELSE statement should be the last CASE statement before END SELECT. It works just like an ELSE in a series of ELSE IF statements.

Here's an example that might appear in a program that reads text, like a compiler or an adventure game.

```
SELECT CASE MID$(LINE$, I, 1)
  CASE "A" TO "Z", "a" TO "z"
    CALL DOWORD(LINE$, I)
  CASE "0" TO "9", "."
    CALL DONUMBER(LINE$, I)
  CASE ELSE
    CALL DOPUNCTUATION(LINE$, I)
END SELECT
```

Finally, if there is no matching CASE statement for a value at all, the program skips to the statement right after END SELECT.

Problem 10.1. Write a program that generates a deck of cards using an array of 52 integers. Initialize the unshuffled deck by placing the numbers 1 to 52 in the array.

Use a subroutine called SHUFFLE to shuffle the deck. This should loop one time through the deck swapping each array element with another chosen at random.

Print the first five cards in the shuffled deck using the PRINTCARD subroutine from this section and a similar subroutine you design to handle printing the suit of the cards.

Just in case your card skills are a little rusty, the names of the suits are Spades, Hearts, Clubs and Diamonds. There are 13 cards in each suit. Card 1 would be the Ace of Spades; card 14 the Ace of Hearts, and so on.

Revisiting the FOR Loop

Once upon a time, in a lesson long, long ago, you learned about the FOR loop. When FOR loops were first introduced, though, you didn't know enough about BASIC to understand some of the features that apply to FOR loops. In this section we will take a more detailed look at FOR loops to fill in some minor gaps in your knowledge.

The first point about FOR loops is one you have seen by example, but it is a good idea to spell it out. You can use any valid BASIC expression to decide what the start and stop value for the loop should be. For example, you can loop a random number of times using the results of the random number function we have used in so many simulations:

```
FOR I = 1 TO RANDOMVALUE (20)
  <<<do something here>>>
NEXT
```

You might be justifiably concerned about what would happen if RANDOMVALUE were called every time the condition was tested. The answer, of course, is that the stop value would change each time through the loop! BASIC evaluates the stop condition one time, though, and saves the value. Even if the stop condition doesn't change, you might be worried about the efficiency of your program. The fact that BASIC computes the stop value before the loop starts, and saves the value, means that even a very complex expression for the stop value won't slow down the loop itself.

There is another interesting point about using an expression for the start or stop value. What happens if the stop value is less than the start value when the loop starts? For example, what does this program do?

```
I = 1
J = -2
FOR K = I TO J
  PRINT K
NEXT
```

The FOR loop can handle this situation. If the stop value is smaller than the start value, the body of the loop is executed one time with the initial value for the loop variable. As soon as the NEXT statement is encountered the loop will stop. This particular program prints the value 1 the first time through the loop, then stops.

So far, all of our FOR loops have started with a small value and looped up towards a larger one. That isn't the only way to loop. You can start with a large value, and loop down to a smaller one. The difference is that you use STEP to set a step size of -1, telling the loop to go down by one each time through the loop rather than up. The program

```
FOR I = 10 TO 1 STEP -1
  PRINT I
NEXT
```

prints a countdown from 10 to 1.

The step size also shows one of the most powerful features of the FOR loop. It isn't limited to INTEGER or even LONG values like the FOR loops in some languages. You can use floating-point loop variables and step by values that are not whole numbers. Here's a short example that uses this fact to step from 0.0 to 2π in increments of $\pi/50.0$. Even if the math is a little beyond what you're used to, you can still see how the FOR loop can be used to loop over non-integer increments.

```
REM Draw 50 random circles on the screen the "hard" way.

DIM I AS INTEGER :! loop variable
DIM R AS INTEGER :! radius of the circle
DIM X, Y AS INTEGER :! position of the center of the circle

CALL INITGRAPHICS
FOR I = 1 TO 50
  R = 10 + RANDOMVALUE(40)
  X = 50 + RANDOMVALUE(220)
  Y = 50 + RANDOMVALUE(100)
  SETSOLIDPENPAT (RANDOMVALUE(15))
  CALL DRAWCIRCLE(X, Y, R)
NEXT
INPUT " ";A$
END
```

```

!-----
!
! DrawCircle - Draw a circle using trigonometry
!
! Parameters:
!   cx, cy - position of the center
!   r - radius
!
!-----

SUB DRAWCIRCLE(CX AS SINGLE , CY AS SINGLE , R AS SINGLE )

CONST PI = 3.1415926535

DIM A AS SINGLE :! for loop angle
DIM X AS SINGLE , Y AS SINGLE :! position on the edge of the
circle

MOVETO (CX + R, CY)
FOR A = 0.0 TO 2 * PI STEP PI / 50.0
  X = CX + R * COS (A)
  Y = CY + R * SIN (A)
  LINETO (X, Y)
NEXT
LINETO (CX + R, CY)
END SUB

!-----
!
! InitGraphics - Set up for graphics
!
!-----

SUB INITGRAPHICS
HGR
SETPENMODE (0)
SETSOLIDPENPAT (15)
END SUB

```

```
!-----  
!  
! RandomValue - Return a random number in the range 1 to max  
!  
! Parameters:  
!   max - maximum allowed value for the random number  
!  
! Returns: Random number in the range 1..max  
!  
!-----  
  
FUNCTION RANDOMVALUE(MAX AS INTEGER ) AS INTEGER  
DIM VALUE AS INTEGER :! Random value to return  
  
VALUE = 1 + RND (1) * MAX  
IF VALUE = MAX + 1 THEN  
    VALUE = MAX  
END IF  
RANDOMVALUE = VALUE  
END FUNCTION
```

There is one other feature of the FOR loop that you won't see in this course, but you might run across in books that show BASIC programs. You can list the loop variable on the NEXT statement, like this:

```
FOR I = 1 TO 10  
    PRINT I  
NEXT I
```

There are two reasons you might want to do this. The first is to give yourself both a comment about which FOR loop the NEXT statement belongs to, and to ask BASIC to check up on you. If you give the wrong FOR loop variable the program will stop with an error.

The other reason to give the name of the FOR loop variable is to tell BASIC to finish two loops with a single NEXT statement. Here's an example that initializes a 10 by 10 matrix with zeros.

```
FOR I = 1 TO 10  
    FOR J = 1 TO 10  
        A(I, J) = 0.0  
    NEXT J, I
```

While this does save one line, I personally think it makes the program harder to read, so I use two separate NEXT statements in situations like this one. It's really a matter of taste, though.

Problem 10.2. One way to reverse a sequence of characters is to loop backwards, starting at the last character in the string, and looping towards the first. Write a program that uses this idea to reverse the characters in a string.

Your program should prompt for a string. Next, print the string in reverse order, using STEP -1 and looping from the length of the string down to 1.

Continue processing strings until the user enters a null string (one with a length of 0).

The GOTO Statement

BASIC became popular on microcomputers before structured programming took hold. Most of the early versions of BASIC did not have modern loop and logic statements like the DO loop, REPEAT loop, or the IF-THEN ELSE statement. Before these statements were available, programmers relied on GOTO statements almost exclusively to control how their programs executed.

The GOTO statement is a jump. The program moves to the destination of the GOTO and starts executing with that statement. The following program gives a very simple example of this idea.

```
GOTO 3
PRINT "This gets skipped."
3 PRINT "This gets printed."
```

As you can see, there isn't much to a GOTO statement. In fact, it's just the reserved word GOTO followed by a number called a label. The number tells the compiler where to go to; a corresponding number must appear somewhere in the program at the beginning of a line.

Modern implementations of BASIC like GSoft BASIC also let you use a name for the label. To use a named label, follow the label name with a colon, like this:

```
GOTO THERE
PRINT "This gets skipped."
THERE: PRINT "This gets printed."
```

The GOTO statement has an interesting history. In a sense, it is a good example of how an idea can be misapplied, abused, and eventually twisted into something the person

who came up with the idea did not intend. What I am referring to, of course, is the idea that GOTO statements are bad. In fact, many people group structured programming and so-called "GOTO-less programming" together, treating them as synonymous. In many computer classes students are still taught that the GOTO statement is always bad. Nothing could be farther from the truth.

In a sense, ignoring the GOTO statement while you learn BASIC is a good idea, up to a point. This is especially true if you learned to program in BASIC or FORTRAN using an older implementation that did not have structured statements like WHILE loops, DO loops and IF-THEN-ELSE statements. Before these statements were available, BASIC programmers had to use IF statements and GOTO statements to do the same thing. That's not altogether a bad thing, but the programs that were written this way tended to jump around seemingly at random, leading to a coding style derisively referred to as spaghetti code. Experience has shown that most programs written using modern statements instead of GOTO statements are easier to read, more efficient, and have fewer bugs than programs written with GOTO statements. So, while you learn the structured statements, and how to use them to organize programs logically, it is a good idea to forget that the GOTO statement exists.

The reason we haven't used the GOTO statement isn't because it is bad, or has no use. The reason we haven't used the GOTO statement is because it isn't needed as much in GSoft BASIC as it is in older versions of BASIC. There are two places, though, where the GOTO statement is very useful, easy to understand, and will make your program much more efficient. These two places are an error exit and an early exit from a loop.

A good example of an early exit from a loop is when you are searching a linked list for a particular item. As a simple example, let's assume that you want to scan a list of names to see if a particular name exists. This problem is a very common one in programming: The list could be a list of names in a customer database, a list of commands that an adventure game recognizes, a dictionary in a spelling checker, or a list of variables in a BASIC program. If the name is in the list, you want to print true. If the name is not in the list, you want to print false.

The ONERR GOTO Statement

Even more important is ONERR GOTO, a variant of the GOTO statement that allows you to intercept errors the BASIC language detects and deal with them on your own terms. ONERR GOTO doesn't actually do anything right away. The line number after the statement is remembered by BASIC, though, and if any error occurs that would normally cause BASIC to stop the program, it jumps to the statement identified in the ONERR GOTO statement instead. You can handle the error there, cleaning up before you exit the program or even handling the error and continuing on.

Here's a short example that shows a complete ONERR GOTO handler. The error itself is something that shouldn't happen in a properly written program—there are better ways to make sure an array subscript isn't out of range than using an ONERR GOTO statement—but this example has the merit of being short.

```
ONERR GOTO 99

DIM A(5) AS INTEGER

FOR I = 0 TO 5
    A(5) = 5
NEXT
I = 7
B = A(I)
PRINT "A(";I;") = ";B
END

99 IF ERR <> 11 THEN
    ONERR GOTO 0
    ERROR ERR
END IF
IF I > 5 THEN
    I = 5
ELSE IF I < 0 THEN
    I = 0
END IF
RESUME
```

Following along as the program executes shows how ONERR GOTO does its job, and also introduces a few commands that you will often use with ONERR GOTO to create an effective error handler.

The ONERR GOTO statement itself doesn't do anything except tell BASIC where to go if an error occurs. If no error is found the program will work exactly the same way with or without the ONERR GOTO statement.

A few lines later the program tries to extract a value from the array A using an index of 7, but the maximum index that is valid for the array is 5. This causes a run-time error, which triggers the error handler. Control jumps immediately to line 99.

The error handler itself shows the three components of a properly written error handler. First the error handler checks to see if the error is something it can handle by checking the value returned by the ERR function. This error handler will only handle error number 11. You can find a list of the errors and error numbers in the GSoft BASIC

reference manual. If the error is not something the error handler can deal with, it uses the statement

```
ONERR GOTO 0
```

to turn off ONERR GOTO handling. Next the error handler *causes* an error using the ERROR statement, which tells BASIC to behave as if a real error was detected. In effect, the error handler has refused to handle any error but error 11, telling BASIC to handle it the way it normally would. Of course, if the program had not turned ONERR GOTO error handling off before doing this, the program would have jumped right back to line 99 to start handling the error again!

The error occurred because the index I was out of range, so the next thing the error handler does is fix the index. Finally, it uses the RESUME statement. This causes the program to go back to the statement that caused the error in the first place and try executing the statement again. If the error occurred inside of a subroutine or function the RESUME statement jumps back to the line in the main program that made the subroutine or function call, not to the line in the subroutine or function that actually generated the error.

You don't have to use the RESUME statement at the end of the error handler. You can use END instead, just like you do at the end of a BASIC program. This lets you stop the program after doing whatever you need to do to handle the error gracefully.

This example is short, but it isn't something that would happen in a real program—or at least not in a well written program. A much better example of a real error you might want to trap is error number 56, a file I/O error. If your program has just modified a critical database or spent hours calculating values for a file, you don't want to lose the information because a disk was full or has a bad block! A properly written error handler can detect this sort of error, giving you a chance to put in a new disk.

Variant Records

We have already seen how records can be used to organize information in our program, grouping any type of variable together into a record about a particular thing. For example, we could use a record to record a person's name, address, and state (all strings), zip code and phone number (possibly integers), and sex. All of these facts about a person can be collected into a single variable, so they can be kept together.

What if we need to keep different information about different groups of people, though? For example, a pet store might want to list whether a fish is a salt-water fish or fresh-water fish, but they certainly wouldn't need to waste space on the same information about a dog. For the dog, they might want to list if it has been spayed or neutered, but the

same information hardly applies to the fish. Rather than waste space by including all of this information when it isn't needed, a variant record can be used.

In a variant record you use a tag variable to keep track of what the record is for. For the pet store, for example, the variant record might look like this:

```
CONST BIRD = 0
CONST FISH = 1
CONST DOG = 2

TYPE ANIMALRECORD
  NEXTP AS POINTER TO ANIMALRECORD
  INSTOCK AS INTEGER
  KIND AS INTEGER
  CASE BIRD
  CASE FISH
    FSEX AS INTEGER
    FRESHWATER AS INTEGER
  CASE DOG
    DSEX AS INTEGER
    SPAYED AS INTEGER
END TYPE
```

There is a wealth of information in this record, so we will take a few moments to study it in detail. The first three variables in the record are NEXTP, INSTOCK and KIND. Up to this point the record looks exactly like any other record, and it is. These three variables are needed no matter what kind of animal we are dealing with, and they will appear in every record of type ANIMALRECORD.

The CASE statement is what makes this record a variant record. The CASE statement looks vaguely like a CASE label in a program, but there are differences. In the variant record, the CASE condition is really just a placeholder. In GSoft BASIC the variable isn't used for anything, although this may change in future versions. It's a good idea to create some constants to record the kind of the record, though, and use the same constants as CASE labels. That's what KIND is for; it will be filled in with BIRD, FISH or DOG to indicate what kind of animal the record refers to.

In this record we decided to record the sex of a FISH or a DOG. Fields in the record must have unique names, even if they appear in different parts of a variant record, so we can't use SEX as the name of both variables. To avoid a conflict, we append a unique letter to the start of the variable names, creating FSEX for the sex of a fish, and DSEX for the sex of a dog. There are other ways to handle the problem of duplicate names, but appending a unique prefix to the field name is a common solution.

Let's take a look at how the same information would be stored in a standard record, and compare the standard record to the variant record. The standard record would look like this:

```
TYPE ANIMALRECORD
  NEXTP AS POINTER TO ANIMALRECORD
  INSTOCK AS INTEGER
  KIND AS INTEGER
  SEX AS INTEGER
  FRESHWATER AS INTEGER
  SPAYED AS INTEGER
END TYPE
```

This record requires 14 bytes of memory: 4 bytes for the pointer (NEXTP), and two bytes for each of the other fields. It also has a FRESHWATER field for birds and dogs, which is not the sort of thing that promotes clarity. The variant record, on the other hand, has a variable size, depending on what kind of animal we are dealing with. In all cases, the size is less than 14 bytes. In the case of a bird, the record has three variables, NEXTP, INSTOCK and KIND. These variables use 8 bytes of memory.

The following example shows one use of variant records. In this example, we create and then animate 10 shapes. The shapes can be squares, triangles, or stars. Each of the shapes does a random walk across the screen, moving one pixel in a random direction on each cycle through the program.

To animate the shapes, we need to keep track of what kind of a shape it is and the coordinates for the shape. Since each shape has a different number of points, we use a variant record. All of the shapes have a color, so that is stored in a non-variant part of the record.

```
REM Do a random walk with 10 random shapes

CONST NUMSHAPES = 10: ! # of shapes to animate
CONST WALKLENGTH = 100: ! # of "steps" in the walk

CONST TRIANGLE = 0: ! shapes
CONST SQUARE = 1
CONST STAR = 2
```

```

! information about one shape
TYPE SHAPERECORD
  COLOR AS INTEGER
  KIND AS INTEGER
  CASE TRIANGLE
    TX1 AS INTEGER
    TX2 AS INTEGER
    TX3 AS INTEGER
    TY1 AS INTEGER
    TY2 AS INTEGER
    TY3 AS INTEGER
  CASE SQUARE
    SX1 AS INTEGER
    SX2 AS INTEGER
    SX3 AS INTEGER
    SX4 AS INTEGER
    SY1 AS INTEGER
    SY2 AS INTEGER
    SY3 AS INTEGER
    SY4 AS INTEGER
  CASE STAR
    PX1 AS INTEGER
    PX2 AS INTEGER
    PX3 AS INTEGER
    PX4 AS INTEGER
    PX5 AS INTEGER
    PY1 AS INTEGER
    PY2 AS INTEGER
    PY3 AS INTEGER
    PY4 AS INTEGER
    PY5 AS INTEGER
END TYPE

DIM I AS INTEGER , J AS INTEGER :! loop variables
DIM SHAPES(NUMSHAPES) AS SHAPERECORD:! current array of shapes
DIM OLDSHAPES(NUMSHAPES) AS SHAPERECORD:! shapes in last
position

! set up the graphics window
CALL INITGRAPHICS
SETPENMODE (2)

```

```
! set up and draw the initial shapes
FOR I = 1 TO NUMSHAPES
  CALL CREATESHAPESHAPES(I)
  CALL DRAWSHAPE(SHAPES(I))
NEXT

! do the random walk
FOR I = 1 TO WALKLENGTH

  ! move the shapes
  FOR J = 1 TO NUMSHAPES
    OLDSHAPES(J) = SHAPES(J)
    CALL UPDATESHAPE(SHAPES(J))
  NEXT

  ! redraw the shapes
  FOR J = 1 TO NUMSHAPES
    CALL DRAWSHAPE(SHAPES(J))
    CALL DRAWSHAPE(OLDSHAPES(J))
  NEXT
NEXT
END
```

```
!-----
!  
! CreateShape - creates a shape  
!  
! The type, color and initial position are chosen randomly.  
! The size of the shape is based on precomputed values.  
!  
! Shared Variables:  
!   triangle, square, star - possible shapes  
!  
! Parameters:  
!   s - shape to create  
!  
!-----
```

```
SUB CREATESHAPESHAPES(S AS SHAPERECORD)
```

```
  SHARED TRIANGLE, SQUARE, STAR
```

```
DIM CX AS INTEGER , CY AS INTEGER :! center point for the
shape
```

```
! get a color
S.COLOR = RANDOMVALUE(15)
```

```
! get the center position, picking the point so the shape is
! on the graphics screen.
CX = RANDOMVALUE(300) + 10
CY = RANDOMVALUE(184) + 8
```

```
! set the initial position
SELECT CASE RANDOMVALUE(3)
```

```
  CASE 1
    S.KIND = TRIANGLE
    S.TX1 = CX - 9
    S.TY1 = CY + 4
    S.TX2 = CX
    S.TY2 = CY - 8
    S.TX3 = CX + 9
    S.TY3 = CY + 4
```

```
  CASE 2
    S.KIND = SQUARE
    S.SX1 = CX - 7
    S.SY1 = CY - 6
    S.SX2 = CX + 7
    S.SY2 = CY - 6
    S.SX3 = CX - 7
    S.SY3 = CY + 6
    S.SX4 = CX + 7
    S.SY4 = CY + 6
```

```
CASE 3
  S.KIND = STAR
  S.PX1 = CX - 6
  S.PY1 = CY + 7
  S.PX2 = CX
  S.PY2 = CY - 8
  S.PX3 = CX + 6
  S.PY3 = CY + 7
  S.PX4 = CX - 10
  S.PY4 = CY - 3
  S.PX5 = CX + 10
  S.PY5 = CY - 3
END SELECT
END SUB

!-----
!
! DrawShape - draw a shape
!
! Shared Variables:
!   triangle, square, star - possible shapes
!
! Parameters:
!   s - shape to draw
!
!-----

SUB DRAWSHAPE(S AS SHAPERECORD)

  SHARED TRIANGLE, SQUARE, STAR

  ! set the pen color for the shape
  SETSOLIDPENPAT (S.COLOR)

  ! draw the shape
  SELECT CASE S.KIND

    CASE TRIANGLE
      MOVETO (S.TX1, S.TY1)
      LINETO (S.TX2, S.TY2)
      LINETO (S.TX3, S.TY3)
      LINETO (S.TX1, S.TY1)
```

```

CASE SQUARE
  MOVETO (S.SX1, S.SY1)
  LINETO (S.SX2, S.SY2)
  LINETO (S.SX4, S.SY4)
  LINETO (S.SX3, S.SY3)
  LINETO (S.SX1, S.SY1)

CASE STAR
  MOVETO (S.PX1, S.PY1)
  LINETO (S.PX2, S.PY2)
  LINETO (S.PX3, S.PY3)
  LINETO (S.PX4, S.PY4)
  LINETO (S.PX5, S.PY5)
  LINETO (S.PX1, S.PY1)
END SELECT
END SUB

!-----
!
! InitGraphics - Set up for graphics
!
!-----

SUB INITGRAPHICS
HGR
SETPENMODE (0)
SETSOLIDPENPAT (15)
END SUB

!-----
!
! RandomValue - Return a random number in the range 1 to max
!
! Parameters:
!   max - maximum allowed value for the random number
!
! Returns: Random number in the range 1..max
!
!-----

FUNCTION RANDOMVALUE(MAX AS INTEGER ) AS INTEGER
DIM VALUE AS INTEGER :! Random value to return

```



```
VALUE = 1 + RND (1) * MAX
IF VALUE = MAX + 1 THEN
    VALUE = MAX
END IF
RANDOMVALUE = VALUE
END FUNCTION
```

```
!-----
!  
! UpdateShape - move the shape across the screen randomly  
!  
! Shared Variables:  
!   triangle, square, star - possible shapes  
!  
! Parameters:  
!   s - shape to update  
!  
!-----
```

```
SUB UPDATESHAPE(S AS SHAPERECORD)
```

```
    SHARED TRIANGLE, SQUARE, STAR
```

```
    DIM DX AS INTEGER , DY AS INTEGER :! movement direction
```

```
    ! get the walk direction  
    DX = RANDOMVALUE(3) - 2  
    DY = RANDOMVALUE(3) - 2
```

! make sure we don't walk off of the screen, then update
! the position

```
SELECT CASE S.KIND
CASE TRIANGLE
  IF DX = - 1 THEN
    IF S.TX1 < 1 THEN
      DX = 0
    END IF
  END IF
  IF DX = 1 THEN
    IF S.TX3 >= 319 THEN
      DX = 0
    END IF
  END IF
  IF DY = - 1 THEN
    IF S.TY2 < 1 THEN
      DY = 0
    END IF
  END IF
  IF DY = 1 THEN
    IF S.TY3 >= 199 THEN
      DY = 0
    END IF
  END IF
```

```
S.TX1 = S.TX1 + DX
S.TY1 = S.TY1 + DY
S.TX2 = S.TX2 + DX
S.TY2 = S.TY2 + DY
S.TX3 = S.TX3 + DX
S.TY3 = S.TY3 + DY
```

```
CASE SQUARE
  IF DX = - 1 THEN
    IF S.SX1 < 1 THEN
      DX = 0
    END IF
  END IF
  IF DX = 1 THEN
    IF S.SX2 >= 319 THEN
      DX = 0
    END IF
  END IF
```

```
IF DY = - 1 THEN
  IF S.SY1 < 1 THEN
    DY = 0
  END IF
END IF
IF DY = 1 THEN
  IF S.SY3 >= 199 THEN
    DY = 0
  END IF
END IF

S.SX1 = S.SX1 + DX
S.SY1 = S.SY1 + DY
S.SX2 = S.SX2 + DX
S.SY2 = S.SY2 + DY
S.SX3 = S.SX3 + DX
S.SY3 = S.SY3 + DY
S.SX4 = S.SX4 + DX
S.SY4 = S.SY4 + DY

CASE STAR
IF DX = - 1 THEN
  IF S.PX4 < 1 THEN
    DX = 0
  END IF
END IF
IF DX = 1 THEN
  IF S.PX5 >= 319 THEN
    DX = 0
  END IF
END IF
IF DY = - 1 THEN
  IF S.PY2 < 1 THEN
    DY = 0
  END IF
END IF
IF DY = 1 THEN
  IF S.PY1 >= 199 THEN
    DY = 0
  END IF
END IF
```

```
S.PX1 = S.PX1 + DX
S.PY1 = S.PY1 + DY
S.PX2 = S.PX2 + DX
S.PY2 = S.PY2 + DY
S.PX3 = S.PX3 + DX
S.PY3 = S.PY3 + DY
S.PX4 = S.PX4 + DX
S.PY4 = S.PY4 + DY
S.PX5 = S.PX5 + DX
S.PY5 = S.PY5 + DY
END SELECT
END SUB
```

Problem 10.3. One common use of variant records takes advantage of the fact that the variables in the variant part overlap. This fact can be used to examine the values of a complicated variable type.

One thing that happens over and over in toolbox programming is to extract the least significant 16 bits from a long, or the most significant 16 bits. You can do this with math operations if you are very careful, but it is much easier and faster to do it with a variant record.

Define a variant record with two variant parts. In one part, define a long integer. In the other part, define two integers, LSW and MSW, in that order. This record puts the two integers in the same memory as the long integer, so that you can save a long value and then extract the integer parts.

Write a program that reads long integers from the keyboard, looping until a 0 is entered. Save this value in the record, then write the two integers.

Experiment with this program a bit. What you should find is that for values up to 32767 the program prints the same value you entered for the least significant integer (the first one), then a zero for the most significant integer (the second one). As the numbers get larger, you start to fill in the sign bit, so the first integer is written as a negative number. Finally, when the numbers exceed 65535, values start to show up in the second integer.

A Quick Tour of Some Advanced GSoft BASIC Features

The next three topics cover some features in GSoft BASIC that are either missing in other implementations of BASIC or are not always implemented the same way. The also are not needed in this course. As you start to write your own programs outside of this

course, though, they are all features you may want to use. The purpose of this section is to make you aware the features exist and show you basically what they are capable of.

Changing the Size of Memory

GSoft BASIC, The FREE Version is limited to 16K of program space and 16K or variable space. The commercial version defaults to 64K for each area. That's more than enough for the programs in this course, and for most other programs, too, but you may eventually write a program that runs out of memory. The SETMEM statement lets you change the amount of memory available in either of these two buffers. See the GSoft BASIC reference manual for details.

Of course, *GSoft BASIC, The FREE Version* doesn't support these commands.

Libraries

There are times in any high-level language where you need to drop into assembly language, either because of speed, space, or very peculiar location requirements for a particular subroutine. GSoft BASIC handles this using libraries, which are also the same as Apple IIGS User Tools.

The commercial version of GSoft BASIC comes with two libraries. We'll use one of those to see how you can use one from your programs. *GSoft BASIC, The FREE Version* doesn't come with any libraries, but it does support them.

The first step in using a library is to make sure it is installed in GSoft BASIC. The two that come with GSoft BASIC are installed when you install GSoft BASIC itself. If you're installing a library yourself, there are two files you need to copy.

First, there will be a file named UserToolxxx, where xxx is a three-digit number from 001 to 255. This file must be copied to the Tools folder in your System folder. The System folder is on the disk you boot from. This disk must also be in the computer when your program runs.

The other file generally has a name like Userxxx.gst, but it can actually have any name at all. The important point is that this file must have a file type of DVU (\$5E) and an auxiliary file type of \$8007. You can put this file several places, including the folder where GSoftBASIC.Sys16 is located or the folder where your program is located.

With the files in place, there are three steps to using a library. First you must load the library with the LOADLIBRARY statement. This actually reads the library from disk and places it in RAM. This is the step where the system folder must be online so the UserToolxxx file can be read from disk. The LOADLIBRARY statement is followed by the number of the library to load; this number is the same as the number that makes up the name of the UserToolxxx file.

The second step is to make calls to the library.

Finally, just before your program exits, it should use the UNLOADLIBRARY statement to remove the library from memory. Like the LOADLIBRARY statement, UNLOADLIBRARY is followed by the number of the library.

Here's a short program that uses the GSoft BASIC time library to read the current date and time.

```
LOADLIBRARY 2
PRINT DATESTRING ;" "; TIMESTRING
UNLOADLIBRARY 2
```

The MakeRuntime Utility

All of the programs you've written so far run from GSoft BASIC itself. That's fine for a program only you use, or while you're developing the program, but once your creation is complete you may want to share it with others that don't have GSoft BASIC. That's where the MakeRuntime utility comes in.

This utility reads the program you create from within GSoft BASIC and creates a program that can run directly from the Finder, even if GSoft BASIC is not installed on the computer. It includes all tool interfaces, your program, the interfaces for any libraries, and enough of GSoft BASIC itself to run your program, cramming all of this into a single file the Finder can execute. The only thing you have to pass on to the person using the program is any libraries you have used. For example, if you wanted to give someone the program from the previous section that prints the date and time, you would also have to give them the file UserTool002 and tell them to copy it to their tools folder.

I won't duplicate the documentation in the reference manual that tells you how to use this utility. The important point is that you know it exists so you can find it when you write a program you want to share.

Lesson Eleven – Scanning Text

The Course of the Course

This lesson, and the three that follow, mark a changing point in the Learn to Program course. Instead of springing it on you with no warning, I thought it would be best to stop and look at what we have done so far and what is left.

The first ten lessons were concerned primarily with teaching you the mechanics of programming. In those lessons you learned most of the features of GSoft BASIC. While we used a number of real programs to illustrate the features of the BASIC language, and frequently discussed principals of good programming practice, programming techniques were not the primary topic.

It turns out that a few tasks turn up repeatedly in many different kinds of programs. The next four lessons deal with some of these basic techniques. In the process, you will get a chance to hone your programming skills.

Because the nature of the material is changing, we will also change our approach a bit. In the first part of the course the text was laced with complete programs to illustrate the basic ideas. As the topics have changed, we have gradually moved away from that technique. Starting with this lesson, we will abandon it almost completely. Instead, we will talk about the concepts behind a particular algorithm. Complete subroutines will be shown in many cases. The problems, for the most part, will involve using these ideas to create complete programs. As always, the solutions are on the disk that comes with the course, so if you get stuck you can always refer to the complete solution.

There are a number of reasons for changing to this approach. One is that you know how to create a program, now, but you still need lots of practice to get really good at it. Another is that we will be able to cover a lot more material this way. Finally, when the course is over, I want you to know how to read intermediate computer science books—the kind of books that teach you about data structures, compiler theory, animation, and so on. Most of these books also give algorithms. If you are used to learning about programming methods by studying algorithms when you see these books for the first time, you will get a lot more out of them. I think it is better to learn to read an algorithm in a setting like this course, when complete programs are at least provided as part of the solution to a problem. In the algorithm books, you won't generally find any complete programs at all.

Manipulating Text

In today's world of graphically based computers, it might seem that manipulating text just isn't important anymore. As it turns out, though, that simply isn't true. Stop and think about it for a moment. The editor you use to type in programs manipulates text. The dialogs you use to enter search strings handle text. The BASIC interpreter that creates programs starts with a text file. From word processors to spread sheets to adventure games, text is still a common way to store information in a computer, so programs still have to manipulate text. That means that, as a programmer, you should know some of the basic techniques used to deal with text.

Programs that deal with text generally divide the task up into well-defined subtasks. These are called scanning, parsing, and semantics. An interpreter is a classic example of a program that manipulates text, so we will start by looking at each of these tasks from the standpoint of a BASIC interpreter. Later, we will see how many other programs use these same ideas.

Scanning, also called lexical analysis, is the process of collecting characters from the text and forming the characters into words. It's not that hard to do, but the idea is a very powerful one. As a quick example, let's look at a simple BASIC program and see how a scanner would break it up into words.

```
PRINT "Hello, world."
```

It is tempting to look at this program as a collection of characters, but if you stop and think about it for a minute, that isn't the way you read it. Instead of individual letters, you group the program into words. BASIC does the same thing. The scanner is responsible for reading the characters and forming words from the characters. These words are called tokens. The main driver for the interpreter never even looks at the characters. Instead, it calls a subroutine, which we will call `NextToken`, that reads characters until a complete word is formed, then returns a single value that indicates what the word is. The scanner would break our short sample program down into reserved words and reserved symbols, like `PRINT`; constants, like the string written by `PRINT`; and identifiers, like the names of any variables. In the case of the identifiers, the scanner also returns a string variable with the name of the identifier. For constants, it returns the value of the constant.

Scanner's aren't limited to interpreters. Virtually any program that deals with words uses a scanner of some sort. Spelling checkers, text adventure games, and even some advanced database programs that accept English-like questions are just a few of the programs that use a scanner.

The next step in the process is called parsing. The parser looks at a sequence of tokens to see if they fit certain preconceived patters. For example, the BASIC interpreter

knows that every line must start with a line number or a command, and if it starts with a line number, the line number must be followed by a command. It has a list of all of the possible commands, and checks this list as it starts to execute a line from the program. Compilers, interpreters, grammar checkers and adventure games are all examples of programs that use parsers.

The last step is called semantic analysis. That's a fancy way of saying that the program figures out what the words mean. In the case of an interpreter, semantic analysis is when the program decides what to do and carries out the task. In an adventure game, semantic analysis is when the game decides that "I want to go north" means that the character should be moved from his current location to another location.

Building a Simple Scanner

The first step in writing a scanner is to decide, in very precise terms, what we mean by a token. In the case of a spelling checker we could define a token as any stream of characters that starts with a letter and contains only letters. Any other characters, such as punctuation marks or numbers, can be ignored, since you can't misspell a number or a comma. You can misuse them, of course, but not misspell them. A BASIC interpreter can't afford to skip commas or numbers, but it can skip comments, spaces, and end of line marks. In other words, one of the jobs of the scanner is to skip characters that are not relevant to the main program.

Let's start with a scanner for a spelling checker. We will skip characters until we get to an alphabetic character, then collect the characters into a string until we get to a non-alphabetic character. We'll break this task down into two parts, reading characters from the file and forming tokens from the characters.

There's more to reading characters from a file than you might think! There are three significant issues to deal with.

First you have to decide how to report the fact that there are no more characters in the file. We'll use a simple but effective way. If there are no more characters in the file, we'll report an empty string for the next character.

The second issue is reporting the end of a line. In programs like a spelling checker we really don't care about the end of the line per se. We do have to do something, though, to handle the situation when one word appears right at the end of a line, and the first character of the following line starts a new word. For our scanner, we'll report the end of the line with a space character.

Finally, we need to read the file efficiently. It may surprise you, but one of the most serious time bottlenecks in every compiler I have ever written is the routine that gets the next character from a file. It's important to make this subroutine work quickly. One of the easiest things we can do in a BASIC program to speed up this process is to read the file in

chunks rather than one character at a time. A convenient chunk in BASIC is a line, so we'll read the file one line at a time. That increases the bookkeeping a bit, but it makes the program a lot faster.

Here's one way to implement the NextCh subroutine, as we'll call it. We'll pick it apart below.

```
!-----  
!  
! NextCh - get the next character from the file  
!  
! Shared Variables:  
!   ch - next character from the file  
!   f - file number  
!   line$ - current line from the file  
!   lineindex - index of the character ch in line$  
!  
! Notes: The end of a line is reported as a space  
!   character.  
!  
!-----  
  
SUB NEXTCH  
  
SHARED CH, F, LINE$, LINEINDEX  
  
! if we need one, get a new line  
IF LINEINDEX > LEN (LINE$) THEN  
  IF EOF (F) THEN  
    CH = ""  
  ELSE  
    LINE INPUT #F, LINE$  
    LINEINDEX = 0  
  END IF  
END IF
```

```
! check for an end of file
IF LEN (CH) <> 0 THEN
  LINEINDEX = LINEINDEX + 1
  IF LINEINDEX > LEN (LINE$) THEN
    ! handle an end of line
    CH = " "
  ELSE
    ! report the next character
    CH = MID$ (LINE$, LINEINDEX, 1)
  END IF
END IF
END SUB
```

The key to understanding how this subroutine works is understanding the variables. As the subroutine runs, it picks characters out of a line read from the input file. The line that we're currently processing is `LINE$`. `LINEINDEX` is the index of the last character we plucked from the line; it will be 0 if we just read a new line. In the normal course of events, the subroutine increments `LINEINDEX` and returns the character at that location in the line.

The first situation that comes up is reaching the end of a line. That's detected right at the start of the subroutine. This section of code also has to handle another exception to the normal flow of events, though, which is reaching the end of the file. If we have reached the end of the file, we set `CH` to an empty string. If we're not there yet the subroutine reads the next line and sets `LINEINDEX` to 0.

The last half of the subroutine reports a character. It starts off with a check to see if we've reached the end of the file, in which case it doesn't need to do anything else. It also checks to see if we've just reached the end of the current line, in which case the subroutine sets `CH` to a space. If we make it past that check, we've handled all possible exceptions to the normal flow of events, so we can return the next character in the current line.

We need to initialize the variables used by `NextCh` before calling in the first time. Here's one way to set them up:

```
CH = " "
LINE$ = ""
LINEINDEX = 0
```

What we've done with these lines is lie to the subroutine, telling it one line has already been read from the file. We've also set `CH` to a space so the end of file check in the second half of the subroutine can't be triggered. The first call to `NextCh` will report a

space as the end of this fake line, so part of the initialization is to call NextCh one time to dump that initial character.

The subroutine NextToken, shown below, breaks the file up into words. While the compares on the IF statements are rather involved, the subroutine itself is actually quite simple. It skips characters, calling NextCh until it finds an alphabetic character or the end of the file. Next it reads characters, appending them to the string TOKEN until a non-alphabetic character or the end of the file is found.

```
!-----  
!  
! NextToken - read a word from the file  
!  
! Shared Variables:  
!   ch - next character from the file  
!   token - string in which to return the token  
!  
!-----  
  
SUB NEXTTOKEN  
  
  SHARED CH, TOKEN  
  
  ! initialize the token  
  TOKEN = ""  
  
  ! skip to the first character  
  WHILE ( ASC (CH) <> 0) AND (CH < "A" OR (CH > "Z" AND CH <  
"a") OR CH > "z")  
    CALL NEXTCH  
  WEND  
  
  ! read the word  
  WHILE ( ASC (CH) <> 0) AND ((CH >= "A" AND CH <= "Z") OR (CH  
>= "a" AND CH <= "z"))  
    TOKEN = TOKEN + CH  
    CALL NEXTCH  
  WEND  
END SUB
```

Problem 11.1. Write a program based on NextCh and NextToken that will scan a text file and write a list of the words in the file, one word per line. As a test, try the program on the source code for the program itself. Be sure you save the program as a source or

text file, though, not a tokenized file. In other words, use the SSAVE or TSAVE command, not the SAVE command, to save the file.

Symbol Tables

One way to write a spelling checker is to collect each word and search for it in a dictionary. Depending on how the spelling checker works, if you find a word that is not in the dictionary, you could print it, display it and let the user correct or accept it, or save it and print a list of words later. This approach works pretty well for interactive spelling checkers. Not so long ago, though, spelling checkers were generally not built right into word processors. Instead, they were separate programs. In this kind of spelling checker, instead of looking up a word as soon as it is found, the words are saved in a linked list. In this kind of spelling checker, only one copy of each word is saved. After the entire document has been scanned, each word is looked up in the dictionary. This drastically cuts the number of times the program needs to look up a word. As a result, the spelling checker is a lot faster than one that looks up each word when it is read from the source file.

This list of words has a name: It is called a symbol table. Finding words in a symbol table is such a common task that an enormous amount of effort has gone into finding very fast ways to look up a word. We'll look at some of these later. For now, though, we will use a simple linked list.

To keep things simple, we generally don't put a word in a symbol table in the NextToken subroutine. Instead, the main program repeatedly calls NextToken, then another subroutine which we will call Insert. Insert creates the symbol table.

In most real programs we put more than just the symbol itself in the symbol table. In our program we will also keep track of how many times the word appeared in the file. The Insert procedure shows how this is done. It uses a record called SYMBOLRECORD, which defines a single entry in the symbol table. This record is defined globally so we can also use a global variable to point to the first element of the linked list. The record looks like this:

```
TYPE SYMBOLRECORD
  NEXTP AS POINTER TO SYMBOLRECORD
  COUNT AS INTEGER
  SYMBOL AS STRING
END TYPE
TYPE SYMBOLPTR AS POINTER TO SYMBOLRECORD
```

You know enough to write Insert on your own. It's job is to scan the symbols already in the symbol table, incrementing the count on the existing symbol if a new word is

already in the table. If the word isn't in the table, Insert should create a new entry in the symbol table for the token and initialize the count to 1. My version is shown in the text.

```
!-----  
!  
! Insert - insert a word in the symbol table  
!  
! Shared Variables:  
!   token - symbol to insert  
!   table - symbol table  
!  
!-----  
  
SUB INSERT  
  
  SHARED TOKEN, TABLE  
  
  DIM SYM AS SYMBOLPTR: ! the symbol we found  
  DIM SPTR AS SYMBOLPTR: ! work pointer  
  
  ! try to find the symbol in the current symbol table  
  SYM = NIL  
  SPTR = TABLE  
  WHILE SPTR <> NIL AND SYM = NIL  
    IF SPTR^.SYMBOL = TOKEN THEN  
      ! yes -> mark the symbol  
      SYM = SPTR  
    END IF  
    SPTR = SPTR^.NEXT  
  WEND  
  
  ! if we didn't find the symbol, create a new one  
  IF SYM = NIL THEN  
    ALLOCATE (SYM)  
    IF SYM <> NIL THEN  
      SYM^.NEXT = TABLE  
      TABLE = SYM  
      SYM^.SYMBOL = TOKEN  
    END IF  
  END IF
```

```
! update the symbol count
IF SYM <> NIL THEN
  SYM^.COUNT = SYM^.COUNT + 1
END IF
END SUB
```

There is one thing about this subroutine that is worth pointing out. What happens if ALLOCATE can't get more memory for a new entry in the symbol table? That's actually very, very unlikely, but assuming "unlikely" is the same thing as "impossible" is one of the easiest ways to create an unreliable program. Sometimes even assuming "impossible" is really impossible can lead to disaster. A classic example is the crash of the first French Arian rocket. Many of the systems in this rocket were from the older version . In that version, a programmer used an integer value for a speed component, knowing the rocket could not go fast enough to overflow the number. You guessed it. The Arian rocket flew fast enough to overflow the buffer, causing it to veer off course, forcing its destruction!

Problem 13.2. Using NextCh, NextToken and Insert, create a program that will count the number of words in a file, and print the number of times each word appears in the file.

Using the techniques covered so far, this program will be very, very slow. Be patient, though. We'll deal with the speed issue later.

Parsing

At one time or another you have probably played one of the adventure games that lets you type text commands. Did you ever wonder how they worked? Some of them can recognize all of these sentences, and in each case they will move the character to the north:

```
Go north.
Run to the north.
I want to move north, now.
North is the direction that I would like to go.
```

Many of these programs are pretty small, so they can't be doing anything particularly difficult. How do they work?

There is one surprisingly simple way to create a program that can recognize and act on all of these commands. It involves building a verb and subject table. Look carefully at the sentences. In each of our examples, there is a verb that indicates you want to move, like go or run. There is also a direction, north. The simple parsers used in the adventure

games scan a sentence looking for a verb and subject the program recognizes. All of the other words are simply discarded. The parser returns the verb and subject, and the program takes some action.

Games aren't the only place this method is used. The same basic idea is used in a program called Eliza, the first computer psychologist. This simple demonstration program is surprisingly effective at giving almost human-like responses, yet it is only a few dozen lines long. An even more direct application of this technology is found in some database query programs written for people who don't normally use computers. For example, you might type

```
Where can I find information  
about Kansas and wheat crops?
```

The database program scans the line, finding just a few relevant words. The verb is find. There are two subjects, Kansas and wheat, separated by a Boolean operator, and. The database program scans its list of articles and books, looking for all of the ones that have both Kansas and wheat in the list of key words.

Let's put these ideas to work in a simple parser to move a spot around on the screen. We are creating a simple robotic control language to move an object around. It would be natural for a person to use a variety of words to describe a direction, and a variety of words to describe movement. For movement, our parser will recognize go and move. For directions, it will recognize left, right, up, down, north, south, east and west. It is the parser's job to make things easy for the main program, so it will report only one value for each direction. We also need a way to quit, so we will add the verb quit to the parser. Quit does not have a subject; it simply means that we are finished. Stop will also be recognized as another form of quit. Our parser assumes that the scanner is converting all characters to uppercase, and that the scanner reads and processes one line at a time, rather than an entire file. In the GetAction subroutine that does the parsing, pay special attention to how NONE is used to indicate that nothing has been found yet. This "empty" value simplifies the program quite a bit.


```
!-----  
!  
! GetAction - find out what the player wants to do  
!  
! Shared Variables:  
!   ch - next character from the file  
!   line$ - line containing the characters  
!   lineindex - index of the character ch in line$  
!   verb - action to take  
!   subject - what we do the action to or with  
!   none, go, quit - verbs  
!   up, down, left, right - subjects  
!   token - string in which to return the token  
!  
!-----  
  
SUB GETACTION  
  
  SHARED CH, LINE$, LINEINDEX, TOKEN  
  SHARED SUBJECT, VERB  
  SHARED NONE, UP, DOWN, LEFT, RIGHT, GO, QUIT  
  
  DIM PROMPT AS STRING :! Prompt for the line input  
  
  ! start with no subject or verb  
  VERB = NONE  
  SUBJECT = NONE  
  
  ! set up a default prompt  
  PROMPT = "Your command, Sir: "  
  
  WHILE VERB = NONE  
    ! get a command line  
    LINE$ = GETLINE(PROMPT)  
  
    ! set up the scanner  
    CH = " "  
    LINEINDEX = 0  
    CALL NEXTCH  
  
    ! handle the command  
    DO  
      ! get the next token  
      CALL NEXTTOKEN
```

```

SELECT CASE TOKEN

    ! handle a subject
CASE "NORTH", "UP":SUBJECT = UP
CASE "SOUTH", "DOWN":SUBJECT = DOWN
CASE "EAST", "RIGHT":SUBJECT = RIGHT
CASE "WEST", "LEFT":SUBJECT = LEFT

    ! handle a verb
CASE "QUIT", "STOP":VERB = QUIT
CASE "GO", "MOVE":VERB = GO
END SELECT
LOOP UNTIL LEN (TOKEN) = 0

! make sure the input is complete and consistent
SELECT CASE VERB
CASE NONE
    PROMPT = "Please tell me what to do (go or stop)."
```

```

CASE GO
    IF SUBJECT = NONE THEN
        PROMPT = "Please tell me which way to go."
        VERB = NONE
    END IF
END SELECT
WEND
END SUB
```

The various values for subjects and verbs, like NONE, DOWN and GO, are declared as constants in the main program.

This is a simple example of a parser. As the number of subjects and verbs increases, the number of rules that are used to combine them also goes up. Some subjects will apply only to certain verbs. In our program, we have an example of a verb, QUIT, that doesn't even have a subject. Some programs also allow subjects with no verb. For example, the adventure game Zork lets you type north, with no verb, to move north. As the possibilities grow, programmers start to use other techniques besides writing SELECT CASE statements for each possibility. Arrays can be used for moderate numbers of subjects and verbs. You index into the array by the subject and verb to find out which subroutine to call. For even more complex programs, techniques for writing rule-based programs can be used. In short, this subroutine gives you some basic ideas you can use to write a program that reads text. If you will be writing large programs using these ideas,

though, you should spend some time looking at the more advanced techniques before starting your program.

There is one interesting problem you will have to deal with to build a program that exercises this parser. It would be natural to draw the robot on the graphics screen, but that leaves us with no way to ask for text input. Mixing text and graphics is pretty tough to do without using the Apple IIGS toolbox, which is beyond the scope of this course. One way to handle this is to draw the robot on the text screen instead of the graphics screen. Creating the robot is easy enough: we can print an asterisk on the screen, erasing it with a space. The problem is positioning the robot on the screen.

There are two statements that make positioning text on the screen fairly easy. HTAB sets the horizontal position where the next character will be printed, and VTAB sets the vertical position. The top left corner on the text screen is 1, 1, with the values incrementing as you move right or down.

Another statement you might find handy in this program is HOME. HOME clears the entire screen and sets the position to 1, 1.

Here's a short subroutine that shows you how you can use these statements to print a character at a specific location on the text screen.

```
!-----  
!  
! DrawRobot - draw the robot  
!  
! Shared Variables:  
!   x,y - position of the robot  
!   ch - character to draw  
!  
!-----  
  
SUB DRAWROBOT(X AS INTEGER , Y AS INTEGER , CH AS STRING )  
  
HTAB X  
VTAB Y  
PRINT CH;  
END SUB
```

Problem 11.3. Write a program to move a spot in the graphics window. The program should use a modified form of the NextToken parser that reads characters from a line instead of a file. NextToken should return tokens with all of the characters converted to uppercase letters.

With these changes in mind, the business end of the main program should include a main loop that looks like this:

```
DO
  ! find out what we are supposed to do
  CALL GETACTION

  ! if it is a movement then move
  IF VERB = GO THEN
    ! erase the old robot
    CALL DRAWROBOT(X, Y, " ")

    ! move the robot
    SELECT CASE SUBJECT
      CASE UP:Y = Y - 1
      CASE DOWN:Y = Y + 1
      CASE LEFT:X = X - 1
      CASE RIGHT:X = X + 1
    END SELECT

    ! draw the robot in the new spot
    CALL DRAWROBOT(X, Y, "**")
  END IF
LOOP UNTIL VERB = QUIT
```

Be sure you remember to initialize X and Y, and draw the starting position of the robot, before the program starts.

GetAction will need to read a line from the user. Read that line from the top of the screen, erasing any old typed input first by writing enough spaces to the screen to clear out the old line.

This problem leaves more of the design of the program to you than any previous problem in the course. If you get stuck, keep in mind that there is a solution on the solutions disk. There are lots of correct ways to write this program. Once you finish, check out the solution to see another way to write the program, and to see where your program seems better organized and what tricks you can pick up from the solution.

Lesson Twelve – Recursion

A Quick Look at Recursion

By now you are well acquainted with defining and calling subroutines. An interesting point about subroutines that we haven't talked about, and that you may not have noticed, is that a subroutine can call itself. The ability of a subroutine to call itself opens up a whole new concept in programming called recursion.

We will start our look at recursion using a simple example. The purpose of this first section is to tell you about the mechanics of recursion. With the mechanics out of the way, we will look at recursion as a problem solving technique, solving the classic problem of the Towers of Hanoi. We will then combine recursion with a simple scanner, like the ones you wrote in the last lesson, to create a recursive descent expression evaluator.

How Procedures Call Themselves

Let's start by looking at a short program. This program multiplies two positive integers.

```
PRINT MULT(4, 5)
END

FUNCTION MULT(X AS INTEGER , Y AS INTEGER ) AS INTEGER
IF Y = 0 THEN
  MULT = 0
ELSE
  MULT = MULT(X, Y - 1) + X
END IF
END FUNCTION
```

Let's face it, that's a pretty weird looking program. We will start by tracing through the program to see how it works.

Stepping through the program, the first thing that happens is MULT gets called with X = 4 and Y = 5. After testing to see if Y is zero, the subroutine executes this statement:

```
MULT = MULT(X, Y - 1) + X
```

This statement is fairly strange all by itself. Here we have a function call, `MULT(X, Y - 1)`, and an assignment to the function. You have seen both of these things by themselves, but never together on the same statement. What does this mean? Well, the statement calls `MULT` again, this time with `X = 4`, and `Y = 4`. Assuming for the moment that this subroutine really will do a multiply properly, the function must return 16. We then add `X`, which is 4, getting an answer of 20. This value is assigned to `MULT`, so it is the value the subroutine will return. Of course, `4*5` is, in fact, 20, so if the call to `MULT` with `X = 4` and `Y = 4` works, the function will actually return the correct answer.

It is fair to ask how the BASIC knows the difference between calling a function and assigning a value for the function to return. After all, the name of the identifier is the same in both cases, and as we have just seen, a function call and an assignment to set the value returned by the function can occur in the same function. The answer lies in which side of the assignment operator the function name is used. If the function name occurs on the left side of the assignment operator, like this:

```
MULT = <some expression>
```

BASIC evaluates the expression on the right side and assigns the value to the function. The function then returns this value to whoever called the function. If the function name is used as part of an expression, like this:

```
<someplace to put the value> = MULT(X, Y - 1) + X
```

BASIC calls the function and uses the value it returns.

Going back to our example, we said that the function would return the correct value if `MULT(4, 4)` returned 16. Convince yourself that it does by tracing through the program. As you go through, assume that `MULT` will return the correct value for `MULT(4, 3)`. You can continue this process right down to the point where a call is made with `Y = 0`.

Problem 12.1. The example showed you how to do a multiplication using recursion. Basically, the program made use of the fact that, when `N` is any number greater than 0, `M * N` gives the same result as `M * (N - 1) + M`. You can find the exponent of a number the same way. For example, `2^3` (2 raised to the power 3) is 8, or `2*2*2`. This is the same as `(2^2)*2`. Change the program so it calculates an exponent, given two integers as input. Use it to verify that `5^4` is 625. As with the addition example, be sure and step through the program.

Recursion is a Way of Thinking

After trying to keep track of all of the values as you traced through a simple recursive program, I don't think it will be hard to convince you that you can't think about recursion the same way you think about IF statements, WHILE loops, and so forth. You will get so tangled up in the details of keeping track of all of the values and how many times the function has been called that you will forget what you are trying to accomplish. You may start to think that anyone that understands recursion must have a mind that would have made Einstein envious. I've watched a number of beginning programmers who would agree as they struggled with recursion, trying to analyze all of those values and calls. It reminds me of the time I opened the course outline for Classical Mechanics in college and saw, on the front page of the outline, in the middle of the page, boldfaced, the following quote:

Any problem, no matter how difficult, can be made still more difficult if looked at it in the right way.

No kidding.

Once you understand that a function can call itself, and that it can have multiple copies of local variables and still keep track of everything, you should never trace through a recursive subroutine, trying to follow the values, again. If you do, you are simply thinking about the problem the wrong way.

Instead, think about a piece of the problem, not the whole thing. Instead of thinking about the multiply as a series of function calls, look at what happens on any particular call. For the multiply function, there are two possibilities: either Y is zero, or it is not. As you know, zero multiplied by any other number is still zero, so we know it is correct for the function to return zero if Y is zero. If Y is not zero, we apply a simple rule: $X * Y$ is the same as $X * (Y - 1) + X$. So, what is $X * (Y - 1)$? We don't know. More important, we don't care. The rule works all of the time, so we truly don't have to worry about what $X * (Y - 1)$ is; a call to a correct multiply routine gives us that answer. With the answer to $X * (Y - 1)$ in hand, we add X and return the correct answer for $X * Y$. The crucial point to remember is that we don't try to trace through the morass of function calls to see what $X * (Y - 1)$ will give us: We recognize that if the function returns the correct value for one terminal case, in our example when $Y = 0$, and that if it returns the correct answer for X and Y, assuming that $X*(Y-1)$ is done correctly, that it must return the correct answer all of the time. Mathematicians call this a proof by induction.

A good way to keep this in mind is to remember that any recursive function must satisfy two conditions to work. First, it has to have a way to stop. In the case of the multiply subroutine, we stopped when Y reached zero. Second, each call must move you

closer to the stopping place than you were when the subroutine was called. In our multiply subroutine, any call that was made with Y greater than 0 reduced Y.

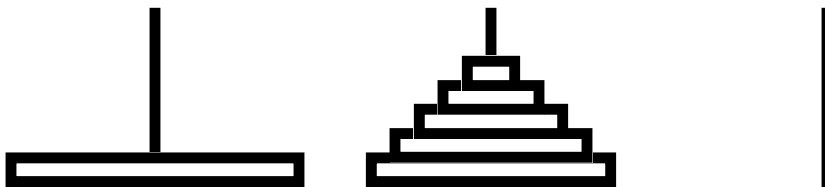
Let's put these ideas to work to solve a classic puzzle, the Towers of Hanoi. This is a puzzle that quickly befuddles anyone who tries to solve it iteratively, the way you have been writing programs up until this lesson. The puzzle starts with six disks, all of a different size, sitting on one of three pegs, like this:



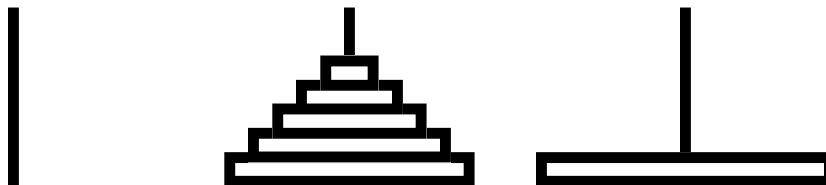
The object is to move all of the disks from the left-hand peg to the right-hand peg. On each turn you can move only one disk. The only other restriction is that you can never cover one disk with a larger disk. Stop and try this before going on. You can cut the six disks from pieces of paper, and stack them on your desk instead of using pegs. You can also do a short version of the puzzle with a penny, nickel, dime and quarter.

So, did you solve the puzzle iteratively? Even if you didn't make any mistakes, it takes 63 different moves to solve the puzzle. Can you keep that many moves straight in your head? If so, you have a better mind than mine.

The way to solve the puzzle is to turn it around. Instead of trying to move the top disk, you have to realize that the real problem is to move the bottom disk! The goal is to move the top five disks from the first peg to the second, like this:



The next step is to move the bottom disk to the third peg.



The last step is to move the pile of five disks from the second peg to the third.



Expressing this as a BASIC procedure, we get something like this:

```
SUB MOVEDISKS(COUNT AS INTEGER , SRC AS INTEGER , DEST AS
INTEGER , SPARE AS INTEGER )

  IF COUNT <> 0 THEN
    CALL MOVEDISKS(COUNT - 1, SRC, SPARE, DEST)
    CALL MOVEONEDISK(SRC, DEST)
    CALL MOVEDISKS(COUNT - 1, SPARE, DEST, SRC)
  END IF
END SUB
```

MOVEONEDISK, of course, is a subroutine that takes the top disk from one peg and places it on another. We could represent the different pegs as three arrays, one for each peg, with six spots in each array. Each spot could be empty, or it might have one of the disks. In practice it's generally easier to have one extra space on each peg that is always empty; this just simplifies the checks that need to be made as you look for the top disk on a peg.

The important thing to recognize is that we haven't worried about how to move five disks from the first peg to the second. We know that if we can move six disks by first moving the top five, then moving the bottom disk, and finally moving the top five disks again, that we can use exactly the same idea to move the five disks. After all, to move five disks, we start by moving four of them to the spare peg, then we move the bottom disk, and finally we move the four disks to the correct peg. To move four disks... well, you get the idea. Eventually, we end up with the trivial problem of moving one disk.

Problem 12.2. Write a program that solves the Towers of Hanoi problem. Draw the disks in the graphics window as they are moved around by the call to MOVEONEDISK.

Problem 12.3. Recursion can be used to process a linked list in reverse order. To see this idea in action, write a program that builds a linked list, stuffing the numbers 1 to 10 in the records, like this:

```
FOR I = 1 TO 10
  ALLOCATE (P)
  IF P <> NIL THEN
    P^.NEXTP = FIRST
    P^.VALUE = I
    FIRST = P
  END IF
NEXT
```

Next, write a recursive procedure that prints the values in the list. On each call, the recursive procedure should return if the pointer that is passed to it is NIL. If the pointer is not NIL, the procedure should call itself, then print the current value, like this:

```
CALL PRINTLIST(P^.NEXTP)
PRINT P^.VALUE
```

After you write the program, reverse the last two statements, and run it again. This time, the program prints the numbers in reverse order. Make sure you understand why, tracing a few iterations with pencil and paper if you really must.

A Practical Application of Recursion

In the last lesson we looked briefly at scanners and parsers. One of the easiest kind of parser to implement is called a recursive descent parser. To see how recursion can be used in a parser, we will solve a problem that had computer scientists stumped for a long time back in the early days of computing, when they were trying to write the first compilers. The problem is to solve a mathematically expressed equation.

For example, you know that

$$(4 + 5) * (1 + 2)$$

is evaluated by adding the terms in parenthesis first, then doing the multiply. How can we write a program that can do this? It's not an idle problem: Over the years I have been asked to write a number of programs that had to solve an equation like this one. The

problem doesn't just crop up in computer languages, either. You need to solve equations in math programs that graph functions, in spread sheets, and even in some databases.

To see how to solve this problem we will write a simple expression evaluator that can add, subtract, multiply and divide. It will accept integer numbers and parenthesis. Just as in algebra and BASIC, add and subtract will have the same precedence, and multiply and divide will have the same precedence, but multiply and divide have a higher precedence than add or subtract.

To get a grasp on how the expression evaluator will work, let's look at this expression:

$$4 * 5 + 9 / 2 - 6$$

To solve this expression by hand we would first scan through, doing all of the multiply and divide operations, leaving only numbers and the add and subtract operations.

$$20 + 4 - 6$$

This equation can be solved by working from left to right, adding and subtracting each new value to the old value. Thinking recursively, we can solve this equation by calling a function to do all of the stuff besides addition and subtraction, then checking to see if there is an add or subtract operation, and finally looping. In true recursive style, not to mention structured programming style, we won't worry about how the subroutine that does the multiplies and divides works. Instead, we solve the smaller problem. Here is our solution, a function that calls another function, FACTOR, to read numbers, do multiplication, and handle parenthesis, does the adds and subtracts that are left over, and returns the result. Our function assumes that the main program calls NEXTTOKEN one time to collect the first token from the input line before expression is called; this is a very common technique in recursive descent parsers.

```
!-----  
!  
! Expression - evaluate an expression  
!  
! Shared Variables:  
!   token - last token read  
!   tokenValue - value of last integer token  
!   t_add, t_subtract, t_integer - names of the tokens  
!  
!-----
```

```

FUNCTION EXPRESSION AS INTEGER

  SHARED TOKEN, TOKENVALUE
  SHARED T_ADD, T_SUBTRACT, T_INTEGER

  DIM FIRSTVALUE AS INTEGER , SECONDVALUE AS INTEGER :! values
from FACTOR
  DIM OPERATION AS INTEGER :! type of the operation

  ! get the first value
  FIRSTVALUE = FACTOR

  ! handle any operations
  WHILE TOKEN = T_ADD OR TOKEN = T_SUBTRACT
    ! skip the operation
    OPERATION = TOKEN
    CALL NEXTTOKEN

    ! get the second value
    SECONDVALUE = FACTOR

    ! do the operation
    IF OPERATION = T_ADD THEN
      FIRSTVALUE = FIRSTVALUE + SECONDVALUE
    ELSE
      FIRSTVALUE = FIRSTVALUE - SECONDVALUE
    END IF
  WEND

  ! return the result
  EXPRESSION = FIRSTVALUE
END FUNCTION

```

Let's trace through this function with our sample expression,

20 + 4 - 6

to see how it works. When the function is called, the main program has already called NEXTTOKEN, so the global variable TOKEN already has a value. It is holding an integer whose value is 20. So far, the function FACTOR doesn't have to do much. It just checks to be sure that token is an integer value, returns the value, and reads in the next

token. When we get to the start of the while loop, then, value is 20. The + character has been read, and TOKEN has been set to ADD.

At the start of the while loop we save the operation in a variable called, surprisingly enough, OPERATION and read the next number. If there is an operation, there must be a number after it. We'll trust FACTOR to flag an error if the number is missing. We then call FACTOR to get the next number, skipping the number token in the process, and do the operation. At the end of the while loop, value is 24, and TOKEN is SUBTRACT. One more pass through the while loop finishes off the expression, and we return a final value of 30.

The next step is to handle multiplication and division. That's no trick, really. They work the same way addition and subtraction do! In this case, we will call a function called TERM to handle numbers and parenthesis. Everything else is an echo of the function that handles addition and subtraction.

```

!-----
!
! Factor - do multiplies and divides
!
! Shared Variables:
!   token - last token read
!   tokenValue - value of last integer token
!   t_multiply, t_divide, t_integer - names of the tokens
!
!-----

FUNCTION FACTOR AS INTEGER

  SHARED TOKEN, TOKENVALUE
  SHARED T_MULTIPLY, T_DIVIDE, T_INTEGER

  DIM FIRSTVALUE AS INTEGER , SECONDVALUE AS INTEGER :! values
from TERM
  DIM OPERATION AS INTEGER :! type of the operation

  ! get the first value
  FIRSTVALUE = TERM

  ! handle any operations
  WHILE TOKEN = T_MULTIPLY OR TOKEN = T_DIVIDE
    ! skip the operation
    OPERATION = TOKEN
    CALL NEXTTOKEN

```

```

! get the second value
SECONDVALUE = TERM

! do the operation
IF OPERATION = T_MULTIPLY THEN
    FIRSTVALUE = FIRSTVALUE * SECONDVALUE
ELSE
    FIRSTVALUE = FIRSTVALUE / SECONDVALUE
END IF
WEND

! return the result
FACTOR = FIRSTVALUE
END FUNCTION

```

Trace through our sample equation

4 * 5 + 9 / 2 - 6

to see how FACTOR works, and how FACTOR and EXPRESSION work together to make sure the operations are done in the correct order. For this short example keeping track of the global variables TERM and TOKEN on a piece of paper should work out well.

The last step is to write the subroutine that handles numbers. There is one other thing that can appear at this point, though, and that is a parenthesis. TERM handles that particular problem by calling EXPRESSION to evaluate whatever appears between the parenthesis! EXPRESSION can then call FACTOR, which will call TERM, and so forth. This recursive call is what allows our expression handler to handle very complex equations.

```

!-----
!
! Term - Handle a number or parenthesis
!
! Shared Variables:
!   token - last token read
!   tokenValue - value of last integer token
!   t_integer, t_lparen, t_rparen - names of the tokens
!
!-----

```

```
FUNCTION TERM AS INTEGER

  SHARED TOKEN, TOKENVALUE
  SHARED T_INTEGER, T_LPAREN, T_RPAREN

  IF TOKEN = T_INTEGER THEN

    ! handle an integer
    TERM = TOKENVALUE
    CALL NEXTTOKEN
  ELSE IF TOKEN = T_LPAREN THEN

    ! skip the (
    CALL NEXTTOKEN

    ! evaluate the expression
    TERM = EXPRESSION

    ! skip the )
    IF TOKEN = T_RPAREN THEN
      CALL NEXTTOKEN
    ELSE
      PRINT "Syntax Error"
    END IF
  END IF
END FUNCTION
```

Take a close look at the error message that is printed if TERM finds an opening parenthesis but no closing parenthesis. Does it look familiar? If not, you might glance through the list of error messages at the end of the GSoft BASIC manual. Now you know where those error messages come from!

Problem 12.4. Write a program to evaluate an expression and write the value. Your program should handle addition, subtraction, multiplication, division, and parenthesis. All operations should be on integers.

Your program should start by prompting the user for an expression. It should then call NEXTTOKEN to fetch the first token from the line, followed by a call to EXPRESSION to evaluate the expression. The program should loop repeatedly, reading new expressions, until the line typed by the user is a null string.

While the text did not cover writing the NEXTTOKEN subroutine, all of the concepts were covered in the last lesson. Try to write NEXTTOKEN on your own; if you get stuck, refer to the solution.

Lesson Thirteen – Sorts

Sorting

Way back in Lesson 5 you got your first look at a sort. Sorting is a pretty common topic in programming courses for a number of reasons. First, there are many places in real programs where you need to sort some information. In some cases, it is pretty obvious that a sort is needed. For example, you may have sorted a database to put a list of people in alphabetical order. You may have sorted the same database to put the list in zip code order to get ready for a mass mailing. In other cases, the fact that something is being sorted is not so obvious, but sorts are none-the-less used. For example, a card playing game may sort a hand of cards.

Another reason sorts are a popular topic is because sorting is a topic that people have spent enough time on to understand fairly well. Computer scientists who deal with the efficiency of algorithms have studied sorts for a long time. In the process, they have compiled a rather impressive list of different ways to sort information.

The Shell Sort

The shell sort is one of several basic sorting methods that are easy to implement, easy to understand, and reasonably efficient for small amounts of information. In the shell sort you loop over the information to be sorted, swapping entries if they are out of order. If you make a swap, you also set a flag to remind you that you found entries that were out of order. In that case, you will need to make another pass over the data to make sure it is in the right order. You keep doing this until you make a pass over the data without finding anything that is out of order. If you are a little fuzzy about the details, refer back to Lesson 5, where this sort was first performed.

Here's a simple version of the sort that sorts an array of `SIZE` numbers, where `SIZE` is a constant or variable telling how many entries are in the array.

```

DO
  SWAP = FALSE
  FOR I = 1 TO SIZE - 1
    IF NUMS[I] > NUMS[I + 1] THEN
      TEMP = NUMS[I]
      NUMS[I] = NUMS[I + 1]
      NUMS[I + 1] = TEMP
      SWAP = TRUE
    END IF
  NEXT
LOOP WHILE SWAP

```

When we start to worry about how efficient a sort is, we usually look at how many times we have to compare the numbers, since that is the operation we do most often. Let's trace through this routine for a short example and find out how efficient it is. We'll use a size of 5, with starting numbers of 5, 4, 3, 2 and 1, in that order. You should follow along with a pencil and paper, writing down the values of variables, executing this algorithm by hand, and counting the operations on your own.

The first time through the loop we do four compares and four swaps. The numbers in the array are ordered like this after the first time through the loop:

4 3 2 1 5

We still have to do four compares each time through the loop. After the next loop, and four more compares, the array looks like this:

3 2 1 4 5

This process continues until the numbers are sorted. We have to do one extra pass after all of the numbers are sorted, since we keep going until SWAP stays FALSE. Here are the numbers in the array, along with the total number of compares we have performed:

2	1	3	4	5	12
1	2	3	4	5	16
1	2	3	4	5	20

While we won't go through a formal mathematical proof, by trying a few cases you can probably convince yourself that if you are sorting N things, and the numbers start out

in reverse order, the number of compares will be $N * (N - 1)$. Starting with the array in reverse order is the worst possible situation for this sort, so we call this the worst case run time.

In a sense, it is pretty unfair to judge anything by the worst case. This is especially true in computer science, since it turns out that in many situations the typical run time for an algorithm is very different than the worst case run time. In fact, there are many situations where the algorithm that has the best worst case run time is not the one with the best typical run time. On the other hand, you do need to know the worst case time, too, since you may be planning a program that is very time critical. In other words, it pays to know as much about algorithms and their efficiency as you can take the time to learn. You may end up picking one method of sorting in one program, and a different method in another.

You will be able to find the worst case run time in published books for most algorithms you are likely to need. What if you can't find out about the algorithm from a book? Or, what if you find the algorithm, but they don't tell you the typical run time, only the worst case run time? Well, you've already seen one way to find the worst case run time, by tracing through the program by hand. You could also do the same thing by machine, of course. While this doesn't give you a mathematical proof, counting the operations does give you a good handle on the run time of an algorithm. You can use the same idea to find the typical run time. These ideas are expanded on in the problems.

Problem 13.1. Write a program that creates an array of integers in reverse order, like the array we looked at in the example in this section. Be sure and use a constant for the size of the array. Sort the array using the algorithm shown, but add a counter that counts the number of compares. Print this value.

Run this program with arrays that have 2, 3, 4, 5, and 10 values. Do all of the numbers match the value $N*(N + 1)$?

Problem 13.2. Finding the typical run time for an algorithm is a lot like finding the worst case run time, like you did in problem 13.1. If you have some actual samples of numbers you plan to sort, you can use the samples to find the typical run time. Another way is to use a simulation, filling the arrays with random values several times, then averaging the run time for the various sorts.

Try this method to find the typical run time for the shell sort. Modify the program from problem 13.1 so it uses a random number generator to fill the array with values between 1 and the size of the array. To keep things simple, allow duplicates. In other words, you don't have to check to be sure that the random number generator returns each possible value once; it is fine if the array has some duplicates. Do this 100 times, and

average the number of compares. Find the values for arrays with 2, 3, 4, 5, and 10 elements.

Quick Sort

There are several ways of sorting information that are a little faster than the shell sort, but these generally still have a run time that is proportional to N^2 , or something pretty close to N^2 , like the $N(N-1)$ that we found for the shell sort. There are also some sorts that have a typical run time proportional to $N \ln(N) / \ln(2)$. To see what this means, let's stop and think about a fairly common sorting problem, sorting a mailing list to zip-code order. There are a variety of mailing lists that come in a variety of sizes, but it isn't uncommon to have 100,000 names in a mailing list. Sorting 100,000 names using the shell sort has a worst case run time of $100,000 \times (100,000 - 1)$, or 9,999,900,000 compares. To say the least, doing nearly ten billion compares takes some serious computer time, especially if you are comparing floating-point numbers, or worse yet, strings. The faster sorts that work in $N \ln(N) / \log(2)$ time, though, would do the same thing using 1,660,964 compares, which is over 6,000 times faster!

The most popular of the fast sorts is a recursive sort called quick sort. Quick sort uses a divide and conquer technique. On each step, a pivot value is picked. Picking a good pivot value is something of a fine art, and it is a very important step. In most cases, the middle value is a good choice for the pivot value. For example, if you are sorting an array with indices from 1 to 100, you would use the 50th element as the pivot value. The routine then moves anything smaller than the pivot value to the left of the pivot, and anything larger than the pivot value to the right of the pivot. The recursive step comes next: the quick sort procedure calls itself, passing the part of the array to the left of the pivot, then makes another recursive call to sort the right half of the array.

Understanding how this works is pretty tricky, so let's get used to it slowly. Type in the following program and make sure it works. It uses quick sort to sort a small array with ten values.

```
REM A sample of quick sort.

CONST SIZE = 10

DIM A(SIZE) AS INTEGER

CALL FILL
CALL SORT(1, (SIZE))
CALL PRINTARRAY
END
```

```
!-----  
!  
! Fill - fill an array  
!  
! Shared Variables  
!   A - array to fill  
!   size - number of elements to fill  
!  
!-----
```

```
SUB FILL
```

```
  SHARED A(), SIZE
```

```
  DIM I AS INTEGER :! loop variable
```

```
  FOR I = 1 TO SIZE
```

```
    A(I) = SIZE + 1 - I
```

```
  NEXT
```

```
END SUB
```

```
!-----  
!  
! PrintArray - print the array  
!  
! Shared Variables  
!   A - array to sort  
!   size - number of elements to fill  
!  
!-----
```

```
SUB PRINTARRAY
```

```
  SHARED A(), SIZE
```

```
  DIM I AS INTEGER :! loop variable
```

```
  FOR I = 1 TO SIZE
```

```
    PRINT A(I)
```

```
  NEXT
```

```
END SUB
```

```

!-----
!
! Sort - sort an array
!
! Shared Variables:
!   A - array to sort
!   size - number of elements to fill
!
! Parameters:
!   left, right - range of indices to sort
!
!-----

SUB SORT(LEFT AS INTEGER , RIGHT AS INTEGER )

  SHARED A(), SIZE

  DIM I AS INTEGER , J AS INTEGER :! array indices
  DIM PIVOT AS INTEGER :! pivot value
  DIM TEMP AS INTEGER :! used to swap values

  ! quit if there is only 1 element to sort
  IF RIGHT > LEFT THEN

    ! find the pivot index
    I = (LEFT - 1) + (RIGHT - LEFT + 1) / 2

    ! put the pivot at the end and save it for compares
    PIVOT = A(I)
    A(I) = A(RIGHT)
    A(RIGHT) = PIVOT

    ! set up the start indices
    I = LEFT
    J = RIGHT - 1

```

```
! partition the array
WHILE I <> J
  WHILE A(I) <= PIVOT AND I <> J
    I = I + 1
  WEND
  WHILE A(J) >= PIVOT AND I <> J
    J = J - 1
  WEND
  TEMP = A(I)
  A(I) = A(J)
  A(J) = TEMP
WEND

! find the pivot insert point
IF A(I) < PIVOT THEN
  I = I + 1
END IF

! replace the pivot
TEMP = A(I)
A(I) = A(RIGHT)
A(RIGHT) = TEMP

! sort to the left of the pivot
CALL SORT(LEFT, I - 1)

! sort to the right of the pivot
CALL SORT(I + 1, RIGHT)
END IF
END SUB
```

Type it in, then run the program once to make sure it is typed in correctly.

For our first look at the `SORT` subroutine, we will not worry too much about how each statement works. Instead, let's look closely at what happens on the whole. The `SORT` subroutine is really divided into four distinct steps:

1. Find a pivot value.
2. Put everything smaller than the pivot to the left of the pivot value, and everything larger than the pivot value to the right of the pivot.
3. Sort the values to the left of the pivot.
4. Sort the values to the right of the pivot.

This is a classic example of recursion as we saw it in the last lesson. To understand quick sort, it is very important to look at what happens on one step, not worrying about how we "sort everything to the left of the pivot."

The first few lines of the procedure find the pivot value and move it to the right-hand side of the array, where it is out of the way:

```
! find the pivot index
I = (LEFT - 1) + (RIGHT - LEFT + 1) / 2

! put the pivot at the end and save it for compares
PIVOT = A(I)
A(I) = A(RIGHT)
A(RIGHT) = PIVOT
```

It may seem strange to go to all of the work to pluck a pivot from the middle of the array and move it to the right-hand side of the array, but there really is a good reason to do this. The algorithm to shuffle the values smaller than the pivot to the left, and the values larger than the pivot to the right, is a lot simpler and faster if we move the pivot value out of the way. It might seem like a good idea to simply use the right-hand value for the pivot, then. It turns out that this is a rotten idea. If you pick the right-hand value for the pivot, and start with a sorted array, quick sort gives the worst performance possible. In practice, picking the middle element of the array for the pivot works very well. An alternate scheme that works even better is to examine the leftmost, rightmost and middle value and pick the middle of the three values. Other schemes for picking a pivot are covered in books that go into more detail on sorting.

The next step is to partition the array. That's the term used to describe the process of shuffling all of the values less of the pivot to the left, and all of the values higher than the pivot right.

```
! set up the start indices
I = LEFT
J = RIGHT - 1

! partition the array
WHILE I <> J
  WHILE A(I) <= PIVOT AND I <> J
    I = I + 1
  WEND
```



```
    WHILE A(J) >= PIVOT AND I <> J
      J = J - 1
    WEND
    TEMP = A(I)
    A(I) = A(J)
    A(J) = TEMP
  WEND
```

This step uses two array indices, I and J. They start at opposite ends of the array, working their way towards the middle until they meet (which means we are finished) or they hit a value that is in the wrong spot. If a value is found that is out of place, it is swapped with another value that is out of place on the other end of the array.

Once the array is partitioned, the pivot value itself is floated to the proper spot in the array.

```
    ! find the pivot insert point
    IF A(I) < PIVOT THEN
      I = I + 1
    END IF

    ! replace the pivot
    TEMP = A(I)
    A(I) = A(RIGHT)
    A(RIGHT) = TEMP
```

With these steps complete, SORT has finished the first cycle through the array and is ready to sort the two partitions. At the point the array actually looks like this:

```
2 3 4 5 1 6 8 9 10 7
```

The partition value of 6 has been floated to its proper spot. Every value smaller than 6 appears to its left, and everything larger than 6 appears to the right.

The last step is to call SORT recursively two times, once for the portion of the array to the left of 6, and once for the portion of the array to the right of 6. Each of these calls will perform this same process to sort the smaller piece of the array until SORT is called with the indexes the same. One element can't be out of order, so that's when SORT finally return without calling itself.

Let's face it: Quick sort is quite a bit more complicated than the shell sort. Why is it faster? After all, if you count the compares in the while loop that partitions the array, we still end up with about N compares. The trick, though, is that quick sort doesn't have to

go through its main loop as many times as the shell sort does. In this example, we've divided the problem in half. Thinking about that in terms of the shell sort, where the worst case sort time is $n*(n-1)$, you can see what an advantage this is. If we are sorting 100 values with the shell sort, the worst case run time is $100*(100-1)$, or 9900. If we sort 2 arrays, each with 50 elements, though, the run time is proportional to $2*(50*(50-1))$, or 4900. You can see that the savings would mount up pretty quickly, since quick sort would divide the 50 element arrays in half, too.

Problem 13.3. How many times does the SORT subroutine get called in the example shown in this section? (Hint: put a counter in the SORT subroutine and run the program.)

Problem 13.4. Find the typical run time for quick sort for arrays that have 2, 3, 4, 5 and 10 elements. Use the same method that you used in problem 13.2. Count the compares of values in the array, but don't count the compares of array indices. There are three places in the subroutine where you will need to increment the counter: inside each of the short WHILE loops, and right after you exit the large WHILE loop.

How do these values compare to the ones you found in problem 13.2?

How Fast Are They?

All of this mathematical gobbledegook about theoretical efficiency may be making your head spin. It can also be taken too far. There are a surprising number of people running around with a degree in computer science who will tell you that quick sort is always faster than a shell sort. Even in theory, this simply isn't true. There are some rare cases where the shell sort will outperform the quick sort if the values in the array happen to be placed just right.

On average, though, quick sort seems like it should work better than the shell sort. It turns out that this isn't quite true. The shell sort has one advantage over quick sort: It is simpler. Recursive subroutine calls take some time; far more time than looping through a WHILE loop. There are also a lot of compares and tests in the quick sort subroutine that aren't needed in the shell sort. It turns out that the shell sort is actually faster than quick sort for small arrays. In fact, on my machine, the solution to problem 13.2 ran faster than the solution to problem 13.4, even though it did more compares. Some sophisticated sorting subroutines take advantage of this fact by using quick sort to sort the array until it is divided into small chunks, then using the shell sort, or one of its close relatives, to sort the small pieces.

This is where practice meets theory. A computer scientist who really understands his topic knows all of this, of course. The theoretical run times are very important, but it is also important to keep the overhead in mind. Unfortunately, while a computer scientist

can use mathematical proofs to find the theoretical run time for an algorithm, there is no easy way to predict the actual run time. That depends on a lot of variables, like how efficient subroutine calls are (they are more efficient compared to loops on an Apple IIGS, for example, than on an IBM 370 mainframe, which does not have a stack), what kind of information you are comparing (integer compares are much faster than string compares), and how long it takes to swap elements of the array (for arrays of records, the swap may take longer than the compare!).

While the theoretical efficiency is a great number to know, there's nothing like actually timing a real program on real data to decide between two competing algorithms.

Quick Sort Can Fail!

One little point has been ignored up to now. Quick sort is very fast, especially for large arrays. Quick sort is a little tougher to implement, but you can modify the SORT subroutine from this lesson fairly easily. The big problem with quick sort is that it doesn't always work.

This may come as quite a shock to you. After all, you stepped through the SORT subroutine fairly carefully. You saw how it worked. How could it fail?

The answer is that there is nothing wrong with the basic idea behind quick sort. Quick sort will always work unless it runs out of memory. You see, every time you make a subroutine call, your program uses a small amount of memory from the variable buffer. The variable buffer is limited in size. By default, programs written in GSoft BASIC have an 64K variable buffer. You can increase this size, but there is always a limit—and the larger the data you're sorting, the less space is left over for subroutine calls.

In version 1.1 of GSoft BASIC, each call to the SORT procedure uses 477 bytes of space from the variable buffer. If you call a subroutine several times from a loop, the procedure uses the same memory each time you call it, but if a subroutine calls itself recursively, each recursive call uses a new chunk of memory. For a variety of reasons, there is no good way to tell in advance exactly how much stack space will be used by a subroutine. Adding a new local variable or switching to a different version of GSoft BASIC will change the memory used. With the default stack size of 64K, and the SORT subroutine we have used in this lesson, it is easy to see that the Sort procedure cannot safely recur more than 137 levels deep. In practice, the value is a little smaller.

If SORT happens to hit a worst-case situation, it will recur as deep as the size of the array. In the best case, Sort will recur $\ln(N)/\ln(2)$ levels deep, where N is the size of the array. This happens when Sort splits the array exactly in half on each call.

All of this points out that you really have to understand not only the advantages of a particular algorithm, but its disadvantages as well. Any algorithm has to be viewed with a critical eye. Quick sort is a lot faster than the shell sort for large arrays, but the shell sort

never fails. And for small array, like the 10 element arrays used in our examples, the shell sort is actually faster than quick sort because it is a less complicated algorithm and ends up executing fewer statements in order to make a swap.

Fortunately, there is a solution to this mess. You can use the `FRE(0)` function to determine how much free memory is left in the variables buffer. If the amount drops below some predetermined limit, say 2000 bytes, you can use a shell sort to sort the piece of the array that you are working on, rather than recurring deeper. You can also time the two sorts for small arrays to find out how large an array needs to be before quick sort is faster, and trigger a shell sort if the array is smaller than that limit.

Problem 13.5. Modify `Sort` so it uses a shell sort if the number of array elements to sort is smaller than `SHELLSIZE`, a constant in your `SORT` procedure. Also, add a constant called `MEMLIMIT` and compare the free memory left in the variables buffer to this value when you enter `SORT`. If the free memory drops below this value, switch to a shell sort. Try your sort on an array with 50 elements.

If you are curious, run the program several times with different values for `SHELLSIZE` to determine the proper value for sorting integers. If you're really curious, you could do the same thing for an array of `DOUBLE` values.

Sorting Summary

Sorting has given you your first real taste of writing efficient programs. You can start to see some of the trade-offs that you will have to make when you write programs, as well as some of the techniques you can use to see the impact of these trade-offs.

You probably know that this lesson has only scratched the surface of sorting. Complete books—long ones, at that—have been written on the topic of sorting. The methods covered in this lesson will work in almost any programming situation you are likely to come across, but if you are ever writing a program that is doing a lot of sorting, it would pay to dig into some books to learn about some of the other sorting methods.

Lesson Fourteen – Searches and Trees

Storing and Accessing Information

The title for this lesson is "Searches and Trees," but a more down-to-earth description would be "better ways to store and find information." Why is this important? Why spend the very last lesson of an introductory programming course on this topic, when there are so many more topics I could have picked?

To answer that, let's step back from the trees a bit and look at the forest. Computers are used for a lot of things, but desktop computers are used most often to display information, make calculations, or store and retrieve information. That's a pretty broad statement, but I think it is true. Spread sheets and engineering calculations are obviously applications where we make calculations. Spread sheets, data bases and spelling checkers are examples of applications where one goal is to store or retrieve information. Word processors, page layout programs, paint programs, and some database programs display information. What about an adventure game, though? Most adventure games are really databases inside, concerned with storing and retrieving information about the adventure world. A chess program is calculation intensive. The list goes on and on.

You already know a few basic ways to store and access information. You have used arrays when you knew how much information would be stored in advance, or when you could put a reasonable limit on the amount of information that would be stored. You have used linked lists when the fixed size of an array created problems. You have even used files when the information had to be written to disk.

This lesson concentrates on two basic themes. If the information is stored in an array, linked list, or disk file, how can you find it quickly? And, what are some better ways to store the information so you can find it even quicker?

Sequential Searches

If you have an array, linked list, or file, the simplest way to find a particular piece of information is to start at the beginning and scan through the data structure until you find the entry you want. This is called a sequential search, and it is nothing new to you. You used a sequential search in Lesson 11 to look for a particular name in a linked list of strings. Of course, you can use a sequential search to look for something in a file or array, too. To look for a numeric value in an array of records, a sequential search would look like this:

```

I = 1
FOUND = FALSE
DO
  IF A(I).AGE = 40 THEN
    FOUND = TRUE
  ELSE
    I = I + 1
  END IF
LOOP UNTIL FOUND OR I = MAXINDEX

```

On average you will have to look through half of the information to find the record you want. If the record doesn't exist—if, for example, you are looking for someone who is 40, but there are no 40 year olds in your data base—you will always scan the entire list. A sequential search, then, has a typical run time of $O(N/2)$ if the item you are looking for is found, and a worst case run time of $O(N)$, where N is the number of things to look at. (The capital O means “on the order of.”)

The Binary Search

The sequential search is a very common kind of search to implement, and it is often the best kind of search to use. In some cases, though, you know more about the information you are searching. For example, one common thing that you might know is that the information is sorted in some kind of order. If you are looking for a man named Smith, for example, you may have ordered your data base so that all of the people are listed in alphabetical order. If you are looking for hospital patients using a Social Security Number, you may be searching a database that is sorted by Social Security Numbers.

When you are searching a list of items that is sorted, and you know in advance how many things are in the array, there is a much better way of finding the information than scanning the array sequentially. The “better way” is called a binary search. The binary search is basically a divide and conquer method, just like quick sort. Binary searches are usually not implemented with recursion, though.

The idea behind a binary search is to start by checking the middle value, rather than the first value. To see how this works, let's assume we are looking for the number 44 in an array of 100 things. The array is very simple: each value is the same as its index, so $A(44)$ is 44. We'll start by looking at the middle value, $A(50)$. The value is 50, which is too large. Since the array is sorted, we know that the value we are looking for must be in the portion of the array from $A(1)$ to $A(49)$, assuming it exists at all. We split the array in half again, and so forth. The table below shows our progress.

index	value	result
50	50	too big
25	25	too small
37	37	too small
43	43	too small
46	46	too big
44	44	match

This divide and conquer search is extremely powerful. Its worst case run time is $O(\ln(N)/\ln(2))$. For our sample of 100 items, a few seconds with a calculator gives the value of 6.64, which tells us that the search will always succeed after no more than 7 compares. That's a big improvement over the sequential sort, with a typical run time for the same array of 50—the binary search is 7 times faster. The larger the array, the bigger the difference, too. For an array with 100,000 values, the sequential search will look at an average of 50,000 values. The binary search will only need to look at 17 values! For an array with 100,000 elements, the binary search is nearly 3,000 times faster.

While there are many twists on the sequential search and binary search, these two basic ideas are at the core of many searches in real programs. Whenever the information you need to search is in no particular order, or is in a linked list, the sequential search is a good choice. If the information is sorted, the binary search is the best choice. Most other searching methods depend on organizing the information better to start with.

Problem 14.1. Develop a binary search algorithm, and test it on a simple array. The search should be implemented as a function that returns the index into the array if the value you pass it is found, and zero if it is not. Use an array of 100 integers, with each array element containing an even number. For example, $A(1)$ would be 2, $A(2)$ is 4, and so forth. Test your search by looking for all of the even numbers from 2 to 200. Make sure the search works when values are not found by passing it 0, 202, and 101.

A Cross Reference Program for BASIC

A binary search is an extremely efficient way of looking for a particular piece of information, but it does have one drawback. While it works well for arrays, it is impossible to implement an efficient binary search for a linked list, simply because you can't hop into the middle of the linked list.

The two most common ways of searching records in dynamically allocated memory are called binary trees and hash tables. Both of these methods use a different way of organizing information to make the search faster. We're going to use a BASIC cross reference program to look at binary trees. A cross-reference program is a program that

looks at the source code to a program and lists all of the places where a particular identifier are used. The purpose of this lesson isn't really to make you write a BASIC cross reference program, so this section gives you one to start with. This BASIC cross reference program uses a linked list for the symbol table.

This program uses the same scanning techniques that we discussed back in Lesson 11, although a few new features have been added to handle comments and to keep track of line numbers. Once a token is found, the program searches for the token in a symbol table that is a simple linked list. If the token does not exist, the search routine creates a new entry in the symbol table. Finally, the program places the line number where the token was found in a linked list in the symbol table. While both the symbol table itself and the line numbers are simple linked lists, this is the first time you have seen a linked list where each element of the linked list point to yet another linked list. There are no new concepts involved in creating linked lists this way, but the details are interesting enough to make it worth looking at the program carefully.

If you have time, you might want to try writing this program on your own before typing in the version you see here.

```
REM XREF
REM
REM This program generates a cross reference of a BASIC
REM program, showing where any symbol is used.

! line number list
TYPE LINERECORD
    NEXTP AS POINTER TO LINERECORD
    NUMBER AS INTEGER
END TYPE
TYPE LINEPTR AS POINTER TO LINERECORD

! symbol table entry
TYPE SYMBOLRECORD
    NEXTP AS POINTER TO SYMBOLRECORD
    SYMBOL AS STRING
    LINES AS LINEPTR
END TYPE
TYPE SYMBOLPTR AS POINTER TO SYMBOLRECORD

CONST F = 1: ! file number
```



```
DIM CH AS STRING :! current character
DIM FNAME AS STRING :! file name
DIM LINE$ AS STRING :! current line
DIM LINEINDEX AS INTEGER :! index into line$
DIM LINENUMBER AS INTEGER :! current line number
DIM SYMBOLS AS SYMBOLPTR:! symbol table
DIM TOKEN AS STRING :! current token
DIM TOKENLINE AS INTEGER :! line number at start of token

! nothing in the symbol table
SYMBOLS = NIL

! first line
LINENUMBER = 0

! get the file name
FNAME = GETFILENAME
IF LEN (FNAME) <> 0 THEN

    ! initialize the scanner
    OPEN FNAME FOR INPUT AS #F
    CH = " "
    LINE$ = ""
    LINEINDEX = 0
    CALL NEXTCH

    ! find all of the symbols
    DO
        CALL NEXTTOKEN
        IF LEN (TOKEN) <> 0 THEN
            CALL INSERT
        END IF
    LOOP UNTIL LEN (TOKEN) = 0

    ! print the symbols
    CALL PRINTSYMBOLS

    ! dispose of the symbol table
    CALL DISPOSESYMBOLS

    ! close the file
    CLOSE #F
END IF
END
```

```
!-----  
!  
! DisposeSymbols - dispose of the symbol table  
!  
! Shared Variables:  
!   symbols - pointer to the first entry in the symbol table  
!  
!-----
```

```
SUB DISPOSESYMBOLS
```

```
  SHARED SYMBOLS
```

```
  DIM LPTR AS LINEPTR:! work line pointer  
  DIM SPTR AS SYMBOLPTR:! work symbol pointer
```

```
  WHILE SYMBOLS <> NIL  
    ! remove the symbol from the symbol table  
    SPTR = SYMBOLS  
    SYMBOLS = SPTR^.NEXTP
```

```
    ! dispose of the lines  
    WHILE SPTR^.LINES <> NIL  
      ! remove the line from the line list  
      LPTR = SPTR^.LINES  
      SPTR^.LINES = LPTR^.NEXTP
```

```
    ! dispose of the line record  
    DISPOSE (LPTR)  
  WEND
```

```
  ! dispose of the symbol record  
  DISPOSE (SPTR)
```

```
WEND
```

```
END SUB
```

```
!-----  
!  
! GetFileName - get the name of the file to cross-reference  
!  
!-----
```

```
FUNCTION GETFILENAME AS STRING

DIM NAME$ AS STRING :! file name

INPUT "File to cross-reference: ";NAME$
GETFILENAME = NAME$
END FUNCTION

!-----
!
! Insert - insert a symbol use in the symbol table
!
! Shared Variables:
!   tokenLine - line number at the start of the token
!   token - symbol to insert
!   symbols - pointer to the first entry in the symbol table
!
!-----

SUB INSERT

SHARED TOKENLINE, TOKEN, SYMBOLS

DIM LPTR AS LINEPTR:! current line number pointer
DIM SPTR AS SYMBOLPTR:! current symbol pointer
DIM WPTR AS SYMBOLPTR:! work symbol pointer

! try to find the symbol
SPTR = NIL
WPTR = SYMBOLS
WHILE WPTR <> NIL
  IF TOKEN = WPTR^.SYMBOL THEN
    SPTR = WPTR
    WPTR = NIL
  ELSE
    WPTR = WPTR^.NEXTP
  END IF
WEND
```

```

! if the symbol isn't in the table then create a new entry
IF SPTR = NIL THEN
  ALLOCATE (SPTR)
  IF SPTR <> NIL THEN
    SPTR^.NEXTTP = SYMBOLS
    SYMBOLS = SPTR
    SPTR^.SYMBOL = TOKEN
    SPTR^.LINES = NIL
  END IF
END IF

! enter the line number
IF SPTR <> NIL THEN
  ALLOCATE (LPTR)
  IF LPTR <> NIL THEN
    LPTR^.NEXTTP = SPTR^.LINES
    SPTR^.LINES = LPTR
    LPTR^.NUMBER = TOKENLINE
  END IF
END IF
END SUB

```

```

!-----
!
! NextCh - get the next character from the file
!
! Shared Variables:
!   ch - next character from the file
!   f - file number
!   line$ - current line from the file
!   lineindex - index of the character ch in line$
!   linenum - current line number
!
! Notes: The end of a line is reported as a space
!        character. All characters are converted to uppercase.
!
!-----

```

```

SUB NEXTCH

```

```

SHARED CH, F, LINE$, LINEINDEX, LINENUMBER

```

```
! if we need one, get a new line
IF LINEINDEX > LEN (LINE$) THEN
  IF EOF (F) THEN
    CH = ""
  ELSE
    LINE INPUT #F, LINE$
    LINEINDEX = 0
    LINENUMBER = LINENUMBER + 1
  END IF
END IF
```

```
! check for an end of file
IF LEN (CH) <> 0 THEN
  LINEINDEX = LINEINDEX + 1
  IF LINEINDEX > LEN (LINE$) THEN
    ! handle an end of line
    CH = " "
  ELSE
    ! report the next character
    CH = MID$ (LINE$, LINEINDEX, 1)
    IF CH >= "a" AND CH <= "z" THEN
      CH = CHR$ ( ASC (CH) - 32)
    END IF
  END IF
END IF
END IF
END SUB
```

```
!-----
!  
! NextToken - read a word from the file  
!  
! Shared Variables:  
!   ch - next character from the file  
!   token - string in which to return the token  
!  
!-----
```

```
SUB NEXTTOKEN
```

```
SHARED CH, TOKEN, TOKENLINE, LINENUMBER
```

```
! initialize the token
TOKEN = ""
```

```

! find the next token
DO
  ! record the line number for the token
  TOKENLINE = LINENUMBER

  IF CH = "!" THEN

    ! handle a comment
    WHILE ASC (CH) <> 0 AND TOKENLINE = LINENUMBER
      CALL NEXTCH
    WEND
  ELSE IF CH >= "0" AND CH <= "9" THEN

    ! handle a number
    WHILE CH >= "0" AND CH <= "9"
      CALL NEXTCH
    WEND
    IF CH = "E" OR CH = "D" THEN
      CALL NEXTCH
      IF CH = "-" OR CH = "+" THEN
        CALL NEXTCH
      END IF
      WHILE CH >= "0" AND CH <= "9"
        CALL NEXTCH
      WEND
    END IF

  ELSE IF ASC (CH) = 34 THEN

    ! handle a string constant
    CALL NEXTCH
    WHILE ASC (CH) <> 34 AND TOKENLINE = LINENUMBER
      CALL NEXTCH
    WEND
    IF ASC (CH) = 34 THEN
      CALL NEXTCH
    END IF

  ELSE IF (CH >= "A" AND CH <= "Z") OR (CH = "_") THEN

```

```
    ! handle a token
    WHILE (CH >= "A" AND CH <= "Z") OR (CH >= "0" AND CH <=
"9") OR (CH = "_")
        TOKEN = TOKEN + CH
        CALL NEXTCH
    WEND
    ELSE IF ASC (CH) <> 0 THEN

        ! handle any other character
        CALL NEXTCH
    END IF
LOOP UNTIL ASC (CH) = 0 OR LEN (TOKEN) <> 0
END SUB
```

```
!-----
!  
! PrintNumber - recursively print the line numbers  
!  
! Parameters:  
!   nPtr - pointer to the remainder of the line number list  
!  
!-----
```

```
SUB PRINTNUMBER(NPTR AS LINEPTR)

IF NPTR <> NIL THEN
    CALL PRINTNUMBER(NPTR^.NEXTP)
    PRINT NPTR^.NUMBER; " ";
END IF
END SUB
```

```
!-----
!  
! PrintSymbols - print the symbols and line numbers  
!  
! Shared Variables:  
!   symbols - pointer to the first entry in the symbol table  
!  
!-----
```

```
SUB PRINTSYMBOLS
```

```

SHARED SYMBOLS

DIM SPTR AS SYMBOLPTR: ! current symbol pointer

SPTR = SYMBOLS
WHILE SPTR <> NIL
    PRINT SPTR^.SYMBOL, ;
    CALL PRINTNUMBER(SPTR^.LINES)
    PRINT
    SPTR = SPTR^.NEXTP
WEND
END SUB

```

There are two ways to save a BASIC program. The SAVE command saves the program as a tokenized file, which replaces BASIC's reserved words with shorter numeric values. This program is designed to process text files, so be sure you try it on programs saved with the SSAVE or TSAVE command. If you want to try to improve the program so it can handle either format, refer to Appendix F of the GSoft BASIC reference manual for details about the tokenized file format used by GSoft BASIC.

Even for text files, though, there are a couple of problems with the BASIC cross reference program you just tried. The program is a lot slower than it could be if we read the file into memory in one chunk using GS/OS disk operating system calls, but the purpose of this lesson isn't learning GS/OS, so we'll put up with that problem. The most subtle problem is that it is a lot slower than it could be, simply because it takes so darn long to deal with a sequential linked list. This is the main problem we will try to solve in the next section. The most obvious problem, though, is that the symbols are printed in the reverse order of when they are first seen in the program. It would be a lot more convenient if they were printed in alphabetical order. We will take care of this problem as a side effect of getting rid of the linked list. The last problem is that any sequence of alphanumeric characters is treated as a symbol. Your program reports all of the places where you used the reserved word end, for example. That one you will solve yourself a bit later, as one of the problems.

The Binary Tree

The major problem with a linked list is the same as the major problem with a sequential search: The program has to scan through an average of half of the list to find a particular entry. If the entry doesn't exist, the program scans through the entire list. A binary tree is another way of handling dynamically allocated records that essentially does the same thing for linked lists that the binary search did for searches. At each level, the tree divides the search in half.

The way this works is to include two pointers to another record in each record, rather than one. In a linked list, each record has a pointer we have called NEXTP that points to the next record in the list. In a binary tree, each record has two pointers, which we will call LEFT and RIGHT. If we look at a particular record, and the one we want is "smaller" than the one we are looking at, we follow the left link. If the one we want is "larger" than the one we are looking at, we follow the right link.

We'll use a few short programs to see how this works. The first task is to learn to add a new item to a binary tree. This is a little harder than it was for a linked list, but the same basic ideas are involved. The program below reads strings from the keyboard and adds them to a binary tree.

```
REM Create a binary tree from keyboard strings

! tree entry
TYPE TREERECORD
  LEFT AS POINTER TO TREERECORD
  RIGHT AS POINTER TO TREERECORD
  STR AS STRING
END TYPE
TYPE TREPTR AS POINTER TO TREERECORD

DIM TREE AS TREPTR: ! top of the tree
DIM TPTR AS TREPTR: ! work pointer
DIM STR AS STRING : ! work string

! nothing in the tree
TREE = NIL

! build the tree of strings
DO
  ! get a string
  LINE INPUT "String: ";STR

  IF LEN (STR) <> 0 THEN
    ! create a new record
    ALLOCATE (TPTR)
    IF TPTR <> NIL THEN
      TPTR^.LEFT = NIL
      TPTR^.RIGHT = NIL
      TPTR^.STR = STR
```

```

        ! add it to the tree
        CALL ADD(TREE, TPTR)
    END IF
END IF
LOOP UNTIL LEN (STR) = 0

! dispose of the tree of string
IF TREE <> NIL THEN
    CALL DISPOSETREE(TREE)
END IF
END

```

```

!-----
!
! Add - add a record to the tree
!
! Parameters:
!   tree - next node in the tree
!   rec - record to add to the tree
!
!-----

```

```

SUB ADD(TREE AS TREEPTR, REC AS TREEPTR)

IF TREE = NIL THEN
    TREE = REC
ELSE IF REC^.STR < TREE^.STR THEN
    CALL ADD(TREE^.LEFT, REC)
ELSE IF REC^.STR > TREE^.STR THEN
    CALL ADD(TREE^.RIGHT, REC)
ELSE
    DISPOSE (TREE)
END IF
END SUB

```

```

!-----
!
! DisposeTree - dispose of the tree
!
! Parameters:
!   tree - node to dispose of
!
!-----

SUB DISPOSETREE(TREE AS TREEPTR)

IF TREE^.RIGHT <> NIL THEN
  CALL DISPOSETREE(TREE^.RIGHT)
END IF
IF TREE^.LEFT <> NIL THEN
  CALL DISPOSETREE(TREE^.LEFT)
END IF
DISPOSE (TREE)
END SUB

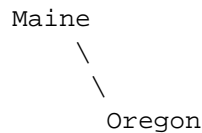
```

One of the first things you might notice as you look at this program is that we are using a recursive subroutine again. Just as with any situation where recursion is useful, we can look at the tree as a piecemeal problem. Let's look at an example to see how this will work. As an example, let's place four states in the tree. We'll use Maine, Oregon, Texas and Colorado for our states. Maine is simple: we create a new record, set LEFT and RIGHT to NIL, record the string, and call ADD. The subroutine ADD sees that TREE is NIL, and records TREE there. The effect on the global variables is to assign PTR to TREE, so tree now points to the first record in our list, Maine. Symbolically, we write the tree like this:

Maine

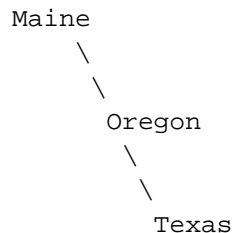
Well, there isn't much there, yet, so our meager tree doesn't look very impressive. Adding Oregon shapes things up a bit, though. This time when we call ADD, the subroutine sees that PTR is not NIL, and checks to see if Oregon is less than Maine. It isn't, so it moves on to the next check to be sure that Oregon is greater than Maine. It is, but let's stop for a moment and consider what would happen if it wasn't. The only way a name could fail both checks is if it matched the name in TREE^.STR exactly. The series of checks, then, prevents duplicates. You can have duplicates in a binary tree, but your search has to take it into account if you do. We don't need them.

At this point, ADD calls itself, passing TREE^.RIGHT as the new top of the tree. TREE^.RIGHT is NIL, so REC is added as the so-called "right child" of Maine. It makes as much sense to call Oregon a branch of Maine, but for historical reasons, we refer to Oregon as the right child of Maine, and Maine as the parent of Oregon. Our tree looks like this, now:

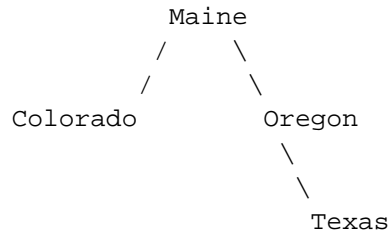


Notice how recursion handled the problem of tracing the tree fairly neatly. Once we decided that the top node existed, and which way to go, we called ADD again, treating TREE^.RIGHT as a brand-new tree, which in a sense it is. If you recall, when recursion was first introduced, I said that the way to think about recursion was to think about one part of the problem at a time. We used that method to solve the Tower of Hanoi problem, where we conceptually moved an entire pile of disks, rather than thinking about the problem as moving individual disks. The same idea cropped up when we used recursion for quick sort, where the subroutine split the problem in half and called itself to solve each half. Here we see the same idea again: ADD decides which half of the tree is the important part, then calls itself, processing the appropriate half of the tree as a new tree.

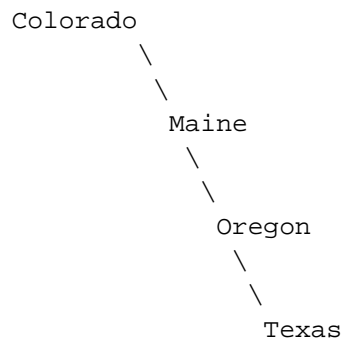
The next state to add is Texas, which makes two recursive calls, getting tacked onto the tree as the right child of Oregon. Follow through the code, writing the steps down on paper if necessary, to see how this is done.



The last state is Colorado. Since Colorado is less than Maine, it is added as the left child of Maine. Our final tree looks like this:



By now, you may have noticed one of the problems with binary trees. To keep the search time to a minimum, you want the tree to be balanced. What that means is that, when you start at the top, the top element of the tree is also the middle element, so that the compare splits the tree in half. In this example, if we had started with Colorado, adding the states in alphabetical order, we would have ended up with a pretty poor excuse for a tree:



You can add a new record to the tree and shuffle the tree around at the same time to make sure it stays balanced. We won't cover how, since it involves some fairly advanced pointer manipulation. In practical situations it often isn't necessary to create a perfectly balanced tree. If records are added to the tree in a fairly random manner the savings of using a tree instead of a linked list are still enormous. Whether the extra effort involved in balancing the tree is worth the time depends on how often the tree will be searched and how random the records are. In our application, they are fairly random.

Searching a binary tree is pretty trivial once you know how to create one. After all, adding a new record to the tree searches the tree as a side effect! Here's a function, based on the ADD procedure, that will search the tree, returning a pointer to the correct record, or NIL if the record does not exist.

```

!-----
!
! Search - search the tree
!
! Parameters:
!   tree - node to search
!   str  - string to look for
!
!-----

FUNCTION SEARCH(TREE AS TREEPTR, STR AS STRING ) AS TREEPTR

IF TREE = NIL THEN
    SEARCH = NIL
ELSE IF TREE^.STR > STR THEN
    SEARCH = SEARCH(TREE^.LEFT, STR)
ELSE IF TREE^.STR < STR THEN
    SEARCH = SEARCH(TREE^.RIGHT, STR)
ELSE
    SEARCH = TREE
END IF
END FUNCTION

```

This is one of those subroutines that you might struggle for a long time to come up with on your own, but is so simple that once you see it, it is easy to understand and remember. Trace through the subroutine, looking for Oregon and Indiana if you aren't sure how it works.

Finally, we come to a subject that impacts directly on our cross-reference program. Using a method called recursive tree traversal we can write a very simple subroutine that will trace through the tree, doing something in order. In our case, we want to print the symbols found in the BASIC program. Here's a simple PRINTTREE subroutine that prints the states in our example program; the subroutine in the BASIC cross reference program will have exactly the same structure.

```

!-----
!
! PrintTree - print the tree
!
! Parameters:
!   tree - tree to print
!
!-----

```

```
SUB PRINTTREE(TREE AS TREEPTR)

IF TREE <> NIL THEN
  CALL PRINTTREE(TREE^.LEFT)
  PRINT TREE^.STR
  CALL PRINTTREE(TREE^.RIGHT)
END IF
END SUB
```

Notice how, once again, recursion simplifies the problem. At any particular place in the tree, we need to print all of the names that come before the one we are working on first, so we call PRINTTREE to do that. Next, we need to print the record we are working on. Finally, we print all of the names that come after the one we just printed. The initial check to make sure TREE is not NIL keeps us from stepping off of the "end" of the tree.

Problem 14.2. Add the print subroutine to the binary tree sample program. Try the program with a variety of names.

Problem 14.3. Change the XREF program so it builds a binary tree for the symbol table instead of a linked list. The easy way to do this is to use the INSERT subroutine to insert each symbol in the program into the symbol table. Because of the way the INSERT subroutine is written, if the symbol already exists, a new symbol is not created. You then call the SEARCH subroutine to find the correct entry in the symbol table (which must exist, since you just created one if there wasn't one already), and enter the appropriate line number.

A more challenging, and more efficient way to implement the program is to combine the Search and Insert subroutines, creating a function that returns a pointer to the correct entry in the symbol table, creating one if one did not already exist. This is the method the solution uses.

In either case, printing the symbol table is a simple matter of modifying the PRINTTREE subroutine from the text.

Problem 14.4. Add a new check to the XREF program that checks to see if the symbol just found is a reserved word in BASIC. You can find a list of the reserved words in Lesson 1.

An easy way to handle reserved words is to add a new flag to each symbol table entry that tells if the entry is a reserved word. If you find a reserved word, you skip adding the

line number to the line number list. When printing the symbol table, you again skip reserved words.

Creating the reserved word list in the first place is a little tedious. You will need a subroutine that calls INSERT for each of the reserved words. There is an optimum order to add the reserved words. See if you can figure it out by thinking about the way trees are created, referring to the example where the names of four states were entered into a tree.

Ruffles and Flourishes

Well, a few weeks ago, you couldn't spell recursive tree traversal, and now you know what it is. Not bad. Let me be the first to congratulate you on joining the ranks of real programmers, who do it with bytes and nibbles.

Of course, as I have pointed out so many times that you may be sick of hearing it, programming is a skill. Like all skills, the more you practice, the easier it gets. There are also a lot more things to learn about programming. Where you go from here depends on your own interests.

BASIC doesn't have a universally accepted standard, but it's generally pretty easy to read books written for any version of BASIC and translate the programs into GSoft BASIC. The exception is books that deal with desktop programming. While the BASIC language won't change enough to make the books impossible to use, the Apple IIGS toolbox is different enough from the way the desktop is implemented on other computers that you won't find much of use from, say a Visual BASIC book.

That leaves an enormous number of good books out there, though. I'd recommend visiting your bookstore, Amazon.com, and especially your local library. While BASIC has seen a resurgence in popularity in the past few years, it's no where near as popular as it was in the early 1980's, when BASIC dominated the microcomputer market. Your library may have a good selection of books from that era on a wide range of topics.

Don't discount other books just because they are written for another language, either. One book I think every programmer should own is

Algorithms

Robert Sedgewick

Addison-Wesley Publishing Company

This is a wonderful encyclopedia of fundamental subroutines that you will use over and over when you program, no matter what computer or language you pick. It was the source for the version of quick sort used in this course, for example.

Algorithms+Data Structures = Programs

Niklaus Wirth

Prentice-Hall

This classic book is a great introduction to intermediate techniques in computer science. It only has five chapters: Fundamental Data Structures, Sorting, Recursive Algorithms, Dynamic Information Structures and Language Structures and Compilers. These chapters give you a basic understanding of data structures that can improve your programming skills enormously. It's written for Pascal, but you should be able to read it and make use of it from GSoft BASIC, too.

If you would like to learn to program the toolbox, writing desktop programs with pull down menus and so forth, you need to study a different set of books. A companion course called *Toolbox Programming in GSoft BASIC* is underway as I write this one. It's designed as a first book for toolbox programming, and comes with it's own abridged toolbox reference manual, so you don't need any other books to get started. For technical references for the toolbox and other parts of the Apple IIGS operating system, see the Byte Works web site, currently hosted at <http://www.hypermall.com/byteworks>.

Whatever you decide to do from here, I hope you enjoyed the course, and learned a few things along the way. Once again, congratulations on completing the course!

Index

! statement, 18

A

addition, 25, 26
ALLOCATE statement, 146
animation, 49–51, 52
arrays, 93–114
 declaring, 93
 multidimensional, 103
 of records, 119
 passing to subroutines, 110
 problems with, 143
 range of indices, 93
ASC function, 84
ASCII character set, 84
assembly language, 178
assignment statement, 18

B

backups, 5
binary operator, 26
binary search, 220–21
binary trees, 229–36
Boolean logic, 55–56
Boolean values, 99
BYTE type, 115

C

CALL statement. *See* subroutines
CASE ELSE statement, 160
case sensitivity, 11
character set, 77, 84–87
CHR\$ function, 84
CLOSE statement, 124, 133

comments, 17, 23, 62
comparing strings, 88
comparisons, 26, 29, 55
compiler, 2
CONST statement, 117
constants, 117
c-string, 87, 88
cursor position, 192

D

DIM statement, 20, 93
 pointers, 144
DISPOSE statement, 146
division, 47
DO-LOOP statement, 38–42
DOUBLE type, 32, 116
double-precision real numbers, 32
drawing
 COPY mode, 50
 exclusive OR mode, 50
dynamic memory, 143

E

EDIT command, 7
ELSE statement. *See* IF statement
END FUNCTION statement. *See*
 subroutines
END IF statement. *See* IF statement
END statement, 62
END SUB statement. *See* subroutines
END TYPE statement. *See* TYPE
 statement
EOF function, 129
ERR function, 167
error handling, 166–68

ERROR statement, 167
errors, 9
evaluating expressions, 200
exclusive OR drawing mode, 50
exponents. *See* real numbers
extended character set, 86

F

false, 55, 99, 118
files, 121–41

- binary files, 133
- closing, 123
- end of file, 129
- file names, 122, 126–29
- file number, 122
- file types, 136
- opening, 122, 133
- opening for input, 124
- path names, 127–28
- random access, 137–41
- writing with PRINT, 123

Finder, 179

FOR statement, 22–23, 161–64

 NEXT, 23, 164

 STEP size, 83, 162

format model, 29

format string, 29

FRE function, 90, 217

FUNCTION statement. *See* subroutines

G

GET statement, 135

GOTO statement, 12, 53, 165–66

graphics, 13–16, 35. *See also* animation
 colors, 16

 drawing a dot, 45

GS/OS, 126

GS/OS strings, 88

GSoft BASIC, The FREE Version!, 5,
 178

H

handling run-time errors, 166–68

HFS disks, 126

HGR statement, 14

HOME statement, 192

HTAB command, 192

I

IF statement, 46–49

 ELSE clause, 48

 ELSE IF clause, 51

 old forms, 53

INPUT statement, 14, 35, 77, 89

INTEGER type, 21, 115

integers, 18, 21, 26, 30, 33

 long. *See* long integers

interpreter, 2

L

LEFT\$ function, 79

LEN function, 79

LET statement. *See* assignment
 statement

lexical analysis, 182

libraries, 178

LINE INPUT statement, 78, 89

line numbers, 12

LINETO tool call, 15

linked lists, 148–56, 199

LIST command, 8

LOADLIBRARY statement, 179

long integers, 27, 33

LONG type, 27, 115

LOOP statement. *See* DO-LOOP
 statement

M

MakeRuntime utility, 179
 memory
 changing size, 178
 memory leak, 152
 MID\$ function, 81
 MOVETO tool call, 15
 multiplication, 19, 25

N

negative numbers, 26
 NIL constant, 150
 null terminated string, 87

O

ONERR GOTO statement, 166–68
 OPEN statement, 122, 124, 133, 137,
 138
 operator precedence, 24
 ORCA shell, 6

P

parenthesis, 26
 parsers, 182, 189–93, 200–205
 pixel, 14
 pointers, 143–56
 declaring, 144
 prerequisites, 4
 PRINT statement, 7, 10, 19
 in files, 123
 PRINT USING statement, 29
 printing, 132
 ProDOS, 126
 ProDOS disks, 126
 p-string, 87
 PUT statement, 134

Q

queues, 155–56
 quick sort, 210–15, 216–17
 QuickDraw II, 14

R

random numbers, 42–46
 real numbers, 18, 21, 27, 30, 33
 exponents, 31
 REAL type, 21
 records, 118–20
 variant, 168–77
 recursion, 195–205
 REM statement, 17
 reserved words, 10, 11
 RESUME statement, 167
 RIGHT\$ function, 79
 RUN command, 8

S

SAVE command, 8
 scanning text, 182, 183–86
 scientific notation. *See* real numbers
 searches, 219–21
 binary search, 220–21
 sequential search, 219
 SELECT CASE statement, 157–61
 semantic analysis, 183
 sequential search, 219
 SETMEM statement, 178
 SETPENMODE tool call, 15
 SETSOLIDPENPAT tool call, 15, 16
 SHARED command. *See* subroutines
 shell sort, 207–9
 SINGLE type, 116
 SIZEOF function, 138
 sorting, 207–18
 quick sort, 210–15, 216–17

- shell sort, 97, 207–9
- stacks, 150–54
- stand-alone programs, 179
- statement separator, 21
- STR\$ function, 89
- STRING type, 21
- strings, 77–91
 - adding, 79
 - ASCII character set, 84
 - character set, 77
 - comparing, 88
 - concatenation, 79
 - constant, 7
 - c-string, 87, 88
 - extended character set, 86
 - garbage collection, 89
 - GS/OS strings, 88
 - null terminated string, 87
 - p-string, 87
 - size limit, 88
 - text blocks, 88
- SUB statement. *See* subroutines
- subroutines, 57–75
 - CALL statement, 60
 - END SUB statement, 61
 - FUNCTION statement, 66
 - parameter list, 60
 - parameters, 69–74
 - passing arrays, 110
 - SHARED command, 74
 - SUB statement, 60
 - value parameters, 70
 - variable parameters, 69
- subtraction, 26
- symbol tables, 168–77

T

- text blocks, 88
- tokens, 182, 183, 186
- true, 55, 99, 118
- truncation, 47
- type characters, 19
- TYPE statement, 117
 - records, 118–20
- types
 - BYTE, 115
 - DOUBLE, 116
 - INTEGER, 115
 - LONG, 115
 - SINGLE, 116
 - UNIV, 115

U

- unary operations, 26
- UNIV type, 115
- UNLOADLIBRARY statement, 179
- User Tools, 178

V

- VAL function, 89
- variable names, 19
- variant records, 168–77
- VTAB command, 192

W

- WEND statement, 28
- WHILE statement, 28, 41