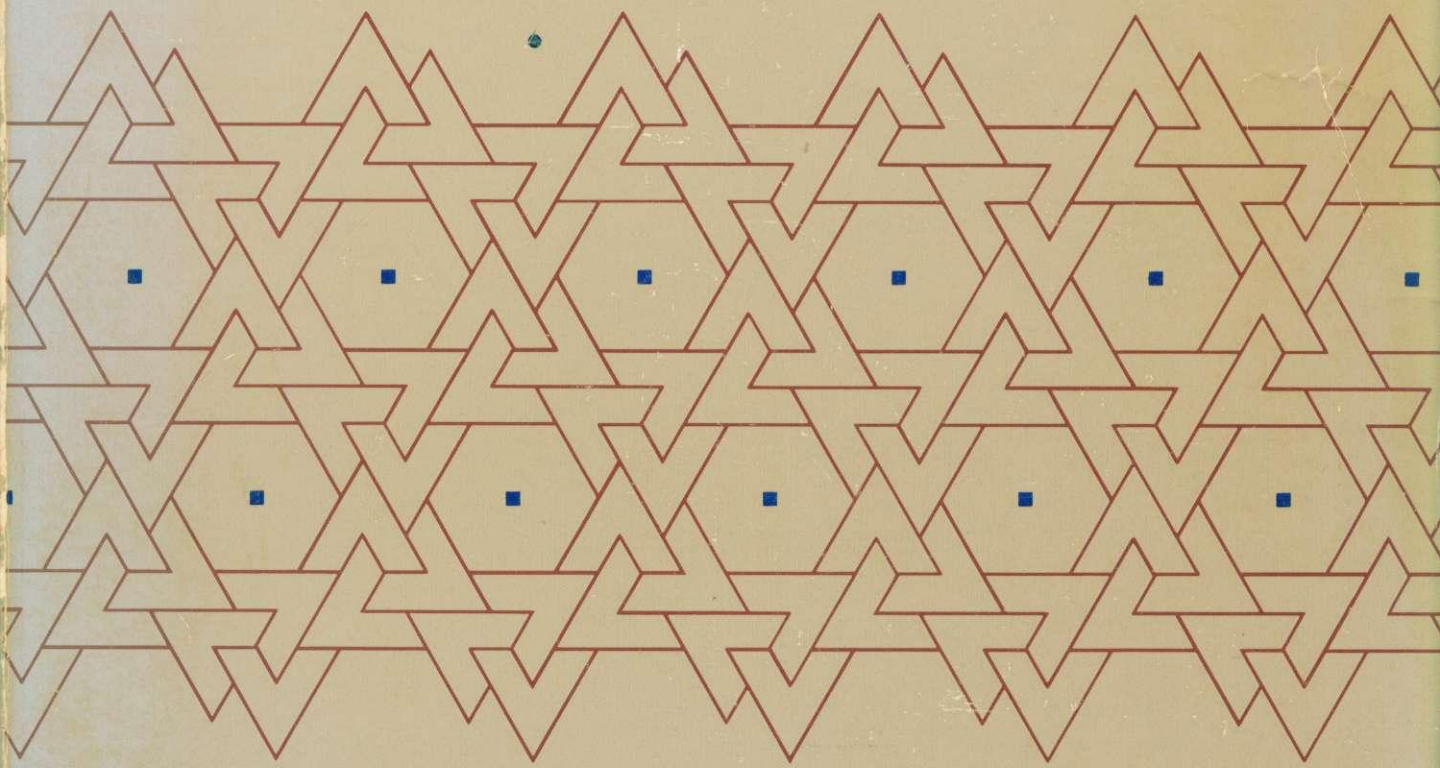
Learning C on the Atari ST

Joseph Boyle Wikert



Scott, Foresman and Company Computer Books

LEARNING C ON THE ATARI ST

LEARNING C ON THE ATARI ST

Joseph Boyle Wikert

Scott, Foresman and Company

Glenview, Illinois *London*

I would like to dedicate this book to my parents, Joseph and Barbara Wikert. Thank you for everything you have done and provided for me in my life.

Library of Congress Cataloging-in-Publication Data

Wikert, Joseph Boyle.
Learning C on the Atari ST / Joseph Boyle Wikert.
p. cm.
Includes index.
ISBN 0-673-18738-1
1. Atari ST computers—Programming. 2. C (Computer program language) I. Title.
QA76.8.A824W55 1988 87-26236
005.265—dc19 CIP

Printer's Key: 1 2 3 4 5 6 EBI 92 91 90 89 88 87

ISBN 0-673-18738-1

Table of Contents

Chapter 1. What Is C? 1

The Origin of C 2
C vs. BASIC and Logo 2
Atari ST C Compilers 3
The Graphics Environment Manager (GEM) 4
Using the Megamax C Compiler 5
Summing Up 12
Glossary for Review 12
Quiz 13

Chapter 2. C Revealed: Structure and Syntax 15

The Right Stuff: Syntax 16
The Elements of a C Program 16
Variables in C 18
The Basic C Data Types 19
Int 19
Float 21
Characters and Strings 22
What to Declare? 24
Assigning Variables 24
Classes of Storage 25
Expressing Ourselves 27
Mathematical Order of Operation 28
Simple C Arithmetic 29

Program Formatting 31
Glossary for Review 32
Quiz 32

Chapter 3. C Statements 33

The printf Function 34
The putchar Function 37
The scanf Function 37
The getchar Functions 39
Statements of Choice: The Conditionals 39
Decisions, Decisions . . . The Thinking ST 40
A Couple of Common Errors with "if" Statements 42
Nesting Your "if" Statements 44
Logical Operators 46
The ?: (Conditional) Operator 46
The Switch Alternative 47
Quiz 49

Chapter 4. More Statements: The Looping Structures 51

Programs That Repeat 52
The "while" Statement 52
Counting Your Loops 54
Loops That Sum 55
More Mathematical Operators 57
The "for" Statement 59
The "do-while" Statement 65
"while" vs. "do-while" 66
Nested Loops 66
One More Loop: The goto Statement 67
Glossary for Review 69
Quiz 69

Chapter 5. C Functions 71

Structured Design 72
What Is a Function? 73
What Does a Function Consist Of? 73
Where Is a Function Placed? 73
How Are Functions Used? 74
Using Variables with Functions 76
Global vs. Local Variables 76
Value Parameters 78
The Return Statement 80
Variable Parameters 82
Glossary for Review 84
Quiz 84

Chapter 6. Library Features 85

How to Avoid Reinventing the Wheel 86
The Central Library 86
Standard C Functions 87
Mathematical 87
String 88
Macros 89
GEM Functions 90
Windows 93
Glossary for Review 97
Quiz 97

Chapter 7. Programmer's Corner 99

The Entire Picture: Complete C Programs 100
Metric Conversions Program 100
Guess a Number Program 104
Decimal to Hexadecimal Conversion Program 110
Tape Counter Program 114
Glossary for Review 117
Quiz 117

Chapter 8. Advanced Data Structures and Concepts 119

Advanced Numeric Types: Longs, Doubles, Words, and Unsigned Types 120
Simple Arrays 121
Parallel Arrays 124
Structures and Files 125
Using Arrays and Structures in an Application 127
Recursion (or, Can I Call Myself?) 130
Glossary for Review 133
Quiz 133

Chapter 9. More on Structures and Files 135

Fields, Structures, and Files 136
Files vs. Arrays of Structures 137
File Handling Library Routines 138
Using a File in a Phone Book Program 139
Sorting and Merging Entries 143
Glossary for Review 150
Quiz 150

Chapter 10. Debugging and File Analysis 151

Debugging 152
File Analysis 153
Glossary for Review 161
Quiz 161

Chapter 11. Graphics 163

Atari ST Graphics in C 164
Fun with the Mouse 171
Glossary for Review 174
Quiz 174

Chapter 12. A Few Programs for the Road 175

The Date Minder Program 176
Batting Average Program 182
Record Album Data Base Program 187
Summing Up 195
Glossary for Review 195
Quiz 195

Appendix A: Reserved Words 197
Appendix B: Answers to Quiz Questions 199
Index 203

Introduction

As the proud owner of an Atari ST personal computer, you are probably already aware of the capabilities of this powerful system. The purpose of this book is to allow you to make your computer do what you want through C, an equally powerful compiler language.

For the beginner, the book starts to teach the basics of C on a very low level. After a few easy-reading chapters, you will be ready to begin writing programs of your own. For the more knowledgeable programmer, special care has been taken to point out unique features of the ST, enabling you to quickly pick up on this revolutionary computer. Finally, when you are finished reading *Learning C on the Atari ST*, you will want to keep it handy as a reference guide.

As you read through the book, you should try out every program that is presented. Enter them into your system, and make sure you understand them thoroughly. After all, this sort of exercise will usually serve to reinforce what you have learned. Good luck, and welcome to the world of C on the Atari ST!

Chapter 1

What Is C?

In this chapter, you will learn:

- A brief history of the C language.
- How C compares with other languages available for the ST.
- What C compilers are available for the ST.
- What GEM is, and how C interfaces with it.
- How to enter, compile, and link a short C program.

The Origin of C

The C language was created by Dennis Ritchie in 1972 at Bell Labs. Since that time, numerous C compilers have been developed to run on various mainframes, mini-, and microcomputers. Most of these compilers claim to be a "Full K & R C Compiler" as one of their selling points. This simply means that their compiler follows the rules of C as laid out in Kernighan and Ritchie's *The C Programming Language* (1978, Prentice-Hall). This book is commonly referred to as the "bible" of C and should be read by anyone who is seriously considering writing in C.

C has become a very popular language over the last several years, mostly because of its power and portability. In most high-level languages (for example, BASIC, PASCAL), the programmer is limited to how close he can get to working with the computer on its own level of bits and bytes. In C, however, you have the ability to manipulate memory all the way down to the bit level, yet the C code is much more readable than the same instructions in assembly language. In addition, because of the Kernighan and Ritchie compatibility, a great deal of code written for one machine may easily be ported to another with few, if any, changes. This point alone is why so many software developers are writing in C; the expense of rewriting a piece of software the size of say, an operating system, in assembly language for another completely different computer can be quite overwhelming. Just imagine writing a sizable program in BASIC on your Atari ST and trying to make "minor" changes to the code so that it will run on your friend's Apple Macintosh. Finally, well-written C code will generally take on a much more compact and concise appearance than the same code written in most other high-level languages. Currently, there are two other popular high-level languages available for the ST: BASIC and Logo.

C vs. BASIC and Logo

When the Atari ST computers were originally introduced, programmers had the option of writing in either BASIC or Logo; interpreters for both these languages are included with the machines. BASIC (Beginner's All-purpose Symbolic Instruction Code) is the result of an attempt to develop a language that was easier to learn than its predecessors. Developed in 1964 at Dartmouth

College, BASIC is the most symbolic of the natural languages. Its instructions are analogous to English-like commands. BASIC has become the most popular language among microcomputer users because of its simplistic approach and ease of use. Logo and BASIC are both used to teach programming to the novice, and Logo, with its unique "turtle" graphics, is geared mostly to younger programmers. BASIC and Logo, however, have their limitations and are not recommended for long, complex programs.

When you program with an interpreter, the code you entered in is translated, or interpreted, into the machine's native language when the program is executed. That is, if you type in the following lines using a BASIC interpreter:

```
10 PRINT "C IS FUN"
20 GOTO 10
RUN
```

the phrase **C IS FUN** will be displayed over and over again until you stop the program. There was no intermediate step in getting your program to run. You simply enter in the statements, and the interpreter acts as somewhat of a hidden middleman by specifying your intentions to the computer.

Using a compiler requires additional step(s), but for the serious programmer, the time spent is well worth the effort. A compiler translates the program into machine-readable form, which may then be stored on a disk or linked into a double-clickable application. This action causes most compiler-generated programs to execute much faster than interpreted ones, since the computer no longer needs to translate your statements one at a time while the program is running.

Now that you understand some of the benefits of using a compiler, specifically C, instead of an interpreter, let's briefly look at a few of the C compilers available for the ST.

Atari ST C Compilers

The author has looked closely at the following three C compilers for the Atari ST computers: Lattice C (Metacomco), Mark Williams C, and Megamax C. Although each of these compilers is fairly easy to use and learn, I recommend the C compiler from Megamax because of its ease of use and performance.

At the time this book was written, Lattice C was the least expensive of the three, and yet the software package itself really has no noticeable shortcomings. One of the major problems with using this C compiler is the lack of support from Metacomco, the company that distributes and supports Lattice C for the Atari ST. Metacomco used to have a phone number in California that users could call for support, but now I understand that you must try and call Great Britain for support—not much of a bargain for an inexpensive compiler. In addition, it appears that both the Mark Williams and Megamax packages are outselling Lattice.

The Mark Williams package (the second most expensive) is very impressive and contains a 600-page manual that thoroughly explains the compiler and most of the standard library routines. My biggest complaint with this compiler is the fact that you really need a hard disk to work with it without driving yourself crazy by swapping floppies. In addition, this is a **five-pass** compiler, whereas Lattice C requires only two passes, and Megamax only one. I also ran into some problems with the **msh** (microshell) utility Mark Williams supplies with the package. One day it would work perfectly fine, and the next day I couldn't compile even the simplest of programs.

Finally, the Megamax C package is by far the easiest and most noteworthy C compiler for the Atari ST I have seen. It boasts an editor that makes excellent use of the mouse and therefore keeps the user from having to remember all the special commands to save, copy, etc. The compile and link processes may be easily combined within the development shell, and the compiler comes with an excellent 442-page manual that thoroughly describes the system.

For these reasons, all the examples in this book are written using Megamax's C, but because of C's portability, very few (if any) modifications to them will allow you to run them using a different compiler. Another reason why you should be able to execute these programs on different compilers is the fact that the Atari uses the Graphics Environment Manager (GEM) to handle a great deal of the processing necessary to take advantage of the ST's powerful capabilities.

The Graphics Environment Manager (GEM)

As the owner of an Atari ST computer, you are already aware of the graphics power this system possesses. The drop-down menus, cute icons, and handy windows are only a few of the responsibilities of the Graphics Environment Manager. This "manager" is actually a collection of efficient routines that

were written for the purpose of making all ST applications "look" and "feel" the same. Just think how confusing it would be if sometimes the menus came from the bottom of the screen up and sometimes from side to side, or if sometimes you had to click twice on an icon to open it, whereas other times it only took one click. If everyone uses these common routines, all programs will operate similarly, and time will be saved.

The GEM library may be further broken down into the Virtual Device Interface (VDI) and the Application Environment System (AES). A description of each GEM routine will be provided as they are used in later chapters. A listing of the specific routines available to you with Megamax C is available in Chapters 11 and 12 of the Megamax manual. The best description of GEM and its routines is provided by the Atari ST GEM Programmer's Reference (published by Abacus Software). This book outlines all of GEM in general, but you will still have to familiarize yourself with the interfaces used by your specific compiler.

Using the Megamax C Compiler

The Megamax C compiler package comes with two $3\frac{1}{2}$-inch microfloppy disks called System and Utility. Briefly, the System disk contains the compiler and linker and all the supplemental files necessary to compile and link a C program, and the Utility disk contains several example programs and a copy of the game Megaroids, which, by the way, is quite fun to play. The author's ST configuration includes two disk drives, but the Megamax C compiler may be run on single drive systems. You may, however, develop a headache having to swap disks unless you can fit your programs on the System disk. For you single drive system owners, any reference to drive B simply means you should eject the current disk and insert the one referred to in the second drive.

To start with, you should place your System disk in drive A and a blank formatted disk in drive B. This blank disk will be used to hold our C source code and executable programs. If you open the window for the System disk, you willl see an icon called SHELL.PRG. This icon represents the driver program of the Megamax system. If you double-click on it, a screen will be displayed with four menus: Desk, File, Execute, and Locate. At this point we are primarily interested in the Execute menu, which contains seven items: Compiler, Linker, Editor, Disassembler, Librarian, Other, and Make. If you select the Editor option, a dialog box is displayed that asks you to select the

file you wish to edit. Select Drive B:, and when the disk stops spinning, select the New box. Another dialog box will be displayed that asks you to enter the name of this new file. Enter the following:

```
welcome.c
```

and press ⟨RETURN⟩ or click the OK box. The editor will be started up. Once it has performed all its setup, the cursor will be blinking in the top left-hand corner of the screen, and you may begin entering C code. The **welcome.c** line that you entered in at the last dialog box means that you wish to open, or in this case create, a file on the disk in drive B by the name of **welcome.c**. "Welcome" is commonly referred to as the **file name** and the **c** portion is called the *extension*. We will always use the **c** extension for files that contain C source code.

Now go ahead and type in the following short C program:

/*First C Program*/

```
#include ⟨stdio.h⟩
main ()
{
printf("Welcome to C");
getchar();
}
```

Just enter the program as if you were typing it into a typewriter. However, the editor is quite a bit more sophisticated than a typewriter; your mistakes can easily be fixed by backspacing over and retyping the incorrect characters. There are several menus in the editor that enable you to enter your C code easily. These are quite well documented in Chapter 5 of the Megamax manual, and it is not the intention of this book to discuss thoroughly any one particular editor. However, most of your code entry can be done simply via the mouse (for positioning the cursor) and the backspace key.

Once you have the program entered, you will want to save it on your source code disk. You can do this by selecting the Save option from the File menu. The disk in drive B will spin for a moment while the file is being saved, and when this process is complete, the editor will be ready for you to enter more text. A great feature of the Megamax editor is that you can have up to four edit windows open at any one time, so you could be looking in different files for routines without having to close one file to get to the next. This feature is also very useful when you are fixing compile errors

because you can open a special file (**errors out**) that contains all your errors and simultaneously have your C source file open for editing.

If you made any errors while entering the program, fix them now, and then use the Save option to update the file. Then, when you have correctly entered the code, select the Quit option in the File menu to leave the editor. Control now is passed back to the shell, and the previous four menus (Desk, File, Execute, and Locate) are displayed. If you were to exit the shell and look at the desktop entry for your disk in Drive B:, you would see an icon for the **welcome.c** file.

At this point, you are ready to compile your first C program. As was mentioned earlier in this chapter, the Megamax C compiler is a one-pass compiler, which means that your programs must go through only one phase of error checking and manipulation before the results are ready to be linked and executed. The compiler may be started up by selecting the Compiler option from the Execute menu. Again, a dialog box is displayed that requests you to enter the name of the file you wish to compile. Select Drive B:, and once the entries for that disk are displayed, either click on the **welcome.c** entry and press the OK box or simply double-click on the **welcome.c** entry. The compilation process will then be started, and you should first see a message that shows the version number of the compiler and one line specifying that the file B:\WELCOME.C is the file being compiled and that the resulting output file is B:WELCOME.o. This latter file is called the **object file,** and it will be used as input for the next phase, linkage. This information is displayed, and assuming no errors were encountered, the shell window is displayed again very quickly. Again, if you were to look at the desktop entry for your disk in Drive B:, you would see an entry for a file called WELCOME.O. This file contains information that is used to link your program and create the final double-clickable application. Before we discuss linking, let's look at what happens if your code contains errors.

Let's say that you accidentally entered the program like this:

```
/*First C Program*/

    #include <stdio.h>
    main()
    {
    printf("Welcome to C")
    getchar();
    }
```

without the semicolon at the end of the **printf("Welcome to C")** statement.

If you then try to compile the program, the compiler will abort with the error:

```
getchar( );
"B:\WELCOME.C", line 7: Function type expected before '('
Number of errors: 1
press RETURN to continue
```

The line number 7 refers to the line number in your source code. If you had a different number of blank lines in your file, your number may be different. After the line number, the error mnemonic is displayed. Although this error doesn't specifically say that you forgot to put a semicolon at the end of your **printf** statement, if you go back and look at your source code, it should be apparent why the compiler got confused. To fix this problem, simply press RETURN, and the Megamax shell will return to your file in the editor. In addition, another window with the file **A:\ERRORS.OUT** is open. As was discussed above, this file contains the error messages encountered in your last compile. You may either select your source code file (by clicking anywhere in it) or close the ERRORS.OUT file by clicking on its close box in the upper left-hand corner. Now go ahead and put the semicolon back on the end of the **printf("Welcome to C")** line, then update and exit the editor via the Save and Quit options in the File menu. When you have returned to the shell, run the compiler (Compiler option in the Execute menu) again on the corrected version of your program so that a valid **WELCOME.o** file will be generated for the linker.

Now that we have an error-free **WELCOME.o** file, the next step in creating a double-clickable application is to link this file with some other files to tie up some loose ends. Our program is quite simple; it will merely display the message **Welcome to C** and wait for you to press a key to continue. Let's take a closer look at what exactly will cause these actions to take place in the code:

```
/*First C Program*/

#include <stdio.h>
main( )
{
printf("Welcome to C");
getchar( );
}
```

The first line of the program is enclosed in the characters /* */. These char-

acters designate a comment, and the compiler ignores everything between them. Comments are very important ways of leaving notes in your code so that you or someone else will know exactly what you were trying to do when you wrote it. The next line in the code instructs the compiler to pull in the file named **stdio.h** (the *.h* extension is used to specify this file as a "header" file) via the **include** directive. Quite simply, when your code is being compiled, if a **#include** is encountered, the compiler treats your program as though the entire file specified was typed in at that spot. If you look at the contents of the **stdio.h** file on your System disk, you'll see it contains a lot of information that, although it may be meaningless to you now, is used to assist the compiler in translating various input/output (I/O) instructions. Input/output instructions are those instructions that allow the computer to accept the information you enter into it and provide information back to you via the screen, speaker, etc. You will see this directive used throughout this book and in most other C programs because 1) it is much easier than typing in the information contained within that file every time you write a program that uses I/O, and 2) even if you did want to have that information in every one of your C source files, if for some reason the **stdio.h** file needs to be changed, you would have to change it in every single program you entered it into!

As you can see, the **#include** directive is a very useful shortcut to reference common external files in your C code without having to worry about entering them in and keeping up with modifications to them.

The next statement in the program is **main( )**. This statement says that this is the start of a routine or function, and this is where the actual program instructions begin. A **function** performs a simple task and is designated by a set of instructions enclosed within the symbols { and }. This particular program has only one function called **main**, but you will later see programs that have numerous other functions used to perform different tasks. Because a program can, and usually does, contain multiple functions, the compiler needs to have some method of determining where to start processing instructions. For this reason, **main** is reserved to designate the beginning of the central routine.

The next statement,

```
printf("Welcome to C");
```

is responsible for displaying the message "Welcome to C" on your screen. **Printf** is an external function that takes parameters (within the parentheses) and, based on certain formatting rules that will be discussed later, sends characters to the monitor. Note the use of the semicolon (;) at the end of this

statement. The compiler must have a method of knowing where each instruction ends and the next one begins, and the semicolon is used for this purpose.

Finally, the last statement enclosed within the main function's start ({) and end (}) symbols is

```
getchar();
```

which is responsible for making the program wait until the user presses the RETURN key on the ST's keyboard. **Getchar** is another external function, and, because of this I/O, we generally need to place the **#include ⟨stdio.h⟩** statement at the beginning of our program. However, the **stdio.h** file does not contain the actual instructions that cause the I/O to work; these instructions are in another compiled file, and the only way we can have them included in our simple program is to use the linker to call them in. This combining of files is performed by the linker, which uses your instructions to tell it exactly which files to work with. It is not important for you to know a great deal about how the linker works, but you should be aware that you can use it to pull together several of your own compiled files as well as other compiled files that come with the Megamax system to create your executable program. Even if you feel you don't need to call in other external files, it is still necessary to put your object file through the linker because the compiler itself does not output a double-clickable application.

With this knowledge about linking, start up the linker by selecting the Linker option from the Execute menu. A new dialog box will be displayed that asks you to keep entering the names of the files you wish to have linked. Select Drive B:, and click on the **WELCOME.O** entry. Then click the ⟩⟩ADD⟩⟩ box, and you should see the entry for **B:\WELCOME.O** in the "To be linked" miniwindow. If we had other files to pull into our link, we could continue to select them in this manner. Now use the keyboard to change the name of the Executable File (at the bottom of the dialog box) from **A.PRG** to **WELCOME.TOS**. Finally, click on the OK button, and the link process will be started.

When the linker starts up, the screen is cleared and information about the linker is displayed. The linker then informs you that your **WELCOME.O** file is being loaded, and then all the external files are traversed to determine if you need any of the information in them. When all the external files have been looked at, some information about the sizes of different segments of your program is displayed, and finally, control is returned to the shell, if no linkage problems are encountered. If an error occurred during linkage, an

error message will be displayed, and you should correct the problem and relink.

After a successful link, you should see an icon named **WELCOME.TOS** on your B disk if you return to the GEM desktop. There's no need to leave the shell to check this, though; simply select the Other option from the Execute menu. In the resulting dialog box, you should see an entry for **WELCOME.TOS**. Now go ahead and double-click on the **WELCOME.TOS** entry. The screen is cleared, the mouse is hidden, and within a few seconds the message **Welcome to C** is displayed on your screen. The message will remain on your screen until you press RETURN. When RETURN is pressed, the program is terminated, and control is returned to the Megamax shell.

An easier method of compiling and linking your programs is provided by Megamax via the "make" utility. Before using "make," however, you must always make sure that system time and date on your ST are current. This may be achieved by using the Control Panel option in the Desk menu. Once you have these items properly set, enter the editor, and create a file called **WELCOME.MAK** on the disk in drive B. Enter these lines in the file:

```
stdio.h
b:welcome.c
mmlink.ttpb:welcome.o-ob:welcome.tos
```

The first line tells "make" that you wish to pull in the **stdio.h** file when compiling. The second line is used to inform the utility that you want to compile the file **welcome.c** on the disk in drive B. Finally, the last line tells "make" that you want to link **b:welcome.o**, with the resulting executable file being **b:welcome.tos**. Notice that your executable program will now be placed on the disk in drive B; this way you can keep your System disk free of unwanted files and keep all your own files on the disk in drive B. Save off the file and open your source code file **b:welcome.c**. The "make" utility checks to see if your source code file has been updated more recently than its associated object file (**b:welcome.o**). If it has been, "make" knows that the file should be recompiled. So even though we have no changes to make to **b:welcome.c**, we must change the time entry for it on the disk so that "make" will recompile it for us. Simply Save and Quit the editor, and select the Make File option from the Locate menu. Select the **WELCOME.MAK** file from drive B and click on the OK button. By doing this, you have informed the

Megamax shell that you want **B:WELCOME.MAK** to be the information file used in the "make" utility. Now select the Make option from the Execute menu, and the compile/link process will be automatically done for you!

Congratulations, you have just finished compiling and linking your first C program on the Atari ST! Go back and review any of the areas that may have been unclear to you on your first reading, and be sure you understand all the concepts presented in this chapter, because they will be the foundation for the remainder of the book.

Summing Up

This chapter has presented you with some basic information on the C language, including a history and comparison of it with the other popular languages available on your Atari ST. A brief discussion of the C compilers available for the ST and the reasons why this book is based on Megamax's product were also discussed. Finally, you have worked with the editor, entered the source code for your first C program, compiled, linked, and executed it, and have been introduced to some of the more fundamental areas of C (comments, the **#include** directive, functions, statements, etc.) as well as Megamax's powerful "make" utility.

Glossary for Review

interpreter—a translation program that converts your code into the machine's native language when the program is executed

compiler—a translation program that converts your code into machine-readable form, which may then be stored on a disk or linked into an application

GEM—Graphics Environment Manager

VDI—Virtual Device Interface

AES—Application Environment System

linker—pulls together one or more compiled files (.o) and creates a double-clickable program

Quiz

1. What does it mean when a compiler is said to be a "full Kernighan and Ritchie C compiler"?
2. What are the two fundamental reasons for C's popularity?
3. How does a C compiler translate this line?

```
/*prigf("Welcome to C");*/
```

Chapter 2

C Revealed: Structure and Syntax

In this chapter, you will learn:

- What the proper structure and syntax of a C program is.
- What the different data types are.
- What a variable is.
- How to use comments and *#define* constants.
- How to use mathematical operations for arithmetic.

This chapter focuses on many programming fundamentals necessary to use C on the Atari ST. The first step in the programming process is to analyze the problem and then to develop a general solution or algorithm. Next, the solution is translated into C language instructions and then implemented to test the results. From now on, we will emphasize the task of writing the program instruction developed from algorithms.

Before we can begin to solve complex problems, we must first master some fundamental concepts of the language itself. The first concept is the structure of a C program and all the elements that make up a simple program. Later in this chapter, we will introduce the concept of a variable and show how this feature is invaluable for more complex programming problems.

The Right Stuff: Syntax

When working with any language, certain rules must be followed. A natural language, such as English, calls these rules grammar. A programming language, like C, refers to these rules as syntax. Writing a program in C requires that the program be properly constructed. Strict rules are imposed when writing programs. Programs must be syntactically correct and use valid statements. If the rules or syntax are not followed—for example, if a semicolon is misplaced or a key word is missing—a program will fail, leaving the programming problem unsolved. Throughout this book, we will learn many syntax rules. In the next section, we introduce a few of the most common ones.

The Elements of a C Program

In general, a C program may be broken up into the following elements (in pseudo–C code):

```
#include files
#define constants
global structures
global variables
main( )
{
```

```
main structures
main variables
main code
}

funct( )
{
func1 structures
func1 variables
func1 code
}

func2( )
{
func2 structures
func2 variables
func2 code
}

funcN( )
{
funcN structures
funcN variables
funcN code
}
```

We saw an example of an **#include** file in Chapter 1 with **#include ⟨stdio.h⟩.** There are, of course, other files that may be pulled into your C source code with the **#include** directive. Any time you see the **#include** directive being used in a program, it simply means that the program needs to reference at least one item specified in that file. As a final note on **#include**, the file name may appear in double quotes (**"filename.h"**) instead of the brackets (⟨**filename.h**⟩). Both methods are correct, but you should choose one and use it always.

The ability to set up symbolic constants in your C programs exists via the **#define** initiative. If you are writing a program that asks users to enter a number, and if it matches your lucky number, they win, you might want to declare your lucky number like this:

```
#define LUCKY 13
```

When your program is compiled, anywhere the characters LUCKY appear (except in quotes), the compiler will replace them with the number 13. In this manner, you can change your lucky number by simply changing its definition in the **#define** statement and not have to worry about changing every occurrence of it in your code. The name LUCKY is called an **identifier** because it identifies the number 13 as a constant that may be used throughout your program.

C places some restrictions on identifiers. For instance, identifiers may only contain upper- or lowercase letters, digits, or the underscore (_) symbol. More specifically, they must begin with a letter, and although they may be several characters in length in Megamax C, only the first ten characters are actually used by the compiler. For this reason, abcdefghijk and abcdefghijl would be considered the same name even though you and I realize the difference. In addition, C programmers have conventionally used uppercase characters for **#define** identifiers and lowercase characters for variable declaration identifiers (which we will discuss shortly). This naming convention makes it easy for a programmer to know immediately whether the name identifies a constant or a variable. You should also note that there is no semicolon used at the end of a **#define** statement.

Both the **#include** and **#define** directives are part of the C preprocessor, which performs various tasks before actually compiling your C program.

The next parts of a C program are the global structures and global variables. In fact, as you can see, the remainder of our C program structure is nothing more than various functions with their appropriate structures and variables. We'll discuss structures shortly, but first let's discuss the concept of variables.

Variables in C

What is a variable? You may recall from basic algebra that a variable is something that you can change. Computers use variables in much the same way: A variable represents something that can change. Specifically, data values are stored in memory at different locations. Each location may contain a value that can change. The computer must have the ability to manipulate data and change its values.

In algebra, a common expression might be represented by $y = x + 2$. The value of y changes as the value of x changes. x and y are called *variables* because you can change their values. Similarly, visualize an empty box marked

X. This box can represent a location in memory, and it can only accept one item at a time. You can put any value in this box you wish, as if it were a constant value, until you change or remove it.

On the Atari ST, each individual box is represented by a numeric address or location. The 520 ST contains approximately 524,000 of these locations in RAM, whereas the 1040 ST contains approximately 1,048,000. Fortunately, C allows us to assign a descriptive name (an identifier) to these locations. Using identifiers to represent specific locations in memory is much easier than using thousands of unique numeric addresses. Each particular identifier must have a type associated with it in order for C to understand how you intend to use it. So let's look at the basic data types available to you in C.

The Basic C Data Types

In programming, data is represented using different types of information, such as letters, numbers, and sets of characters. Before these types of information can be processed, they first must be defined. In this section, we review the common data types required for short programs. These data types include int, float, and char.

Int

An *int* (short for integer) is a whole number, which may be either positive or negative, that cannot include a comma. Signs—positive (+) and negative (−)—are allowed. If a positive sign is omitted, the positive is assumed. As you know, the number of integers is infinite. However, there is a limit to the number available on the ST in Megamax C. The largest number represented by an int type is 32767, while the smallest is −32768. In any of our variable sections listed above, we could insert the line:

```
int a;
```

and we would then have a space in memory set aside that could be referenced by the variable name **a** and could contain any of the values between and including 32767 and −32768. An int requires 16 bits of storage space in Megamax C and therefore is two bytes long. A *byte* is simply a group of 8

bits, or binary digits, which is used to hold data on a computer in the base 2, or binary, number system. For those of you unfamiliar with binary math, the following is a brief table of decimal numbers and their binary equivalent:

Decimal	*Binary*
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101

It is not necessary that you thoroughly understand the binary numbering system now, but if you are unfamiliar with it, it is suggested that you read one of the numerous mathematics books that are dedicated to this system.

There are other data types that may be used to define an integer besides the int. If you do not need the full two bytes of storage for your variable, you could declare **a** to be a short int like this:

```
short int a;
```

or, more simply:

```
short a;
```

Either method will define a variable named **a** that occupies one byte or 8 bits in memory, half the size of a regular int. Alternatively, we could define **a** as a long int like this:

```
long int a;
```

or, again leaving off the int descriptor:

```
long a;
```

The long int in Megamax C on the ST defines a storage location of 32 bits or 4 bytes, twice the size of the normal int declaration.

Another modifier, **unsigned,** may be used to alter the range of values allowed in your int variables. An unsigned short, declared like this:

```
unsigned short a;
```

still allocates one byte of storage, but now the range of allowable values in **a** is 0 to 255. An unsigned int may be declared like this:

```
unsigned a;
```

which results in the allocation of 16 bits of storage in which any of the values 0 to 65535 may reside. An unsigned long int may be declared like this:

```
unsigned long a;
```

which results in the allocation of 32 bits of storage in which any of the values 0 to 4294967295 may reside. In summary, Megamax C's various int data types on the ST are:

Data Type	*Size in Bytes*	*Range*
int	2	−32768 to 32767
long int	4	−2147483648 to 2147483647
unsigned int	2	0 to 65535
unsigned long int	4	0 to 4294967295
short int	1	−128 to 127
unsigned short int	1	0 to 255

Notice that the use of the unsigned modifier does not affect the storage size allocation, but does affect the range of values that may occupy that space. As you know, mathematical operations are not limited to whole number values. For this reason, the float data type may be used to work with real numbers.

Float

Real numbers are numbers that contain decimal numbers. Generally, at least one digit is placed to the right of the decimal point, indicating the fractional part of the real number. In C, real numbers are called **floats** because of their floating decimal point, which is not restricted to any particular position. Floats are commonly referenced in what is called scientific or exponential notation.

Scientific notation is the method the ST uses to work with very large

or very small numbers. For example, the number 5 million, represented as 5.0×10^7, is written as:

```
5.0E + 7
```

where the number 5.0 is called the *coefficient* or *mantissa,* and the number +7 is called the *exponent*. The 5 is the whole part of the number to the left of the decimal point, while the fractional part, or exponent, represents the power of ten or the number of places to the right of the decimal point. The letter "E" means exponent. Therefore, to determine a number written in scientific notation, move the decimal point of the mantissa the number of places specified by the exponent to the right. Another example: 4.43E + 2 represents 443.

Similarly, very small numbers are represented with scientific notation. For example, the real number represented by .00001234 is written as:

```
1.234E - 5
```

where the negative exponent means that the decimal point of the mantissa should be moved five spaces to the left instead of to the right. For a negative real number, the negation sign is placed in front of the mantissa; that is, −4.52E − 3 represents −.00452.

In Megamax C, the data types float and double, or long float, are available for variable definition. The declaration

```
float a;
```

defines a variable space that is 32 bits long and may hold any of the values from +/−10E−37 to +/−10E38. A double float is declared like this:

```
double a;
```

in which a represents a float that is 64 bits long and may be in the range of +/−10E−307 to +/−10E308.

Characters and Strings

Fortunately, numbers are not the only data types that can be manipulated with C. We can also work with text information that is represented by upper- and

lowercase letters, special characters, etc. Such information is represented by the data type *char*. A char variable may be declared like this:

```
char a;
```

which would result in the allocation of one byte of memory for a character value. Thus, **a** could hold the value 'M', or '8', or '#', etc. When we show a character value, it is always represented with that character surrounded by single quotes (' '). At this point you can see that the char data type is quite useful. But what makes it even more powerful is the fact that it can be used to define variables that hold more than one character—called strings. A *string* is nothing more than a collection of characters with a known end point. We set up a char variable to hold more than one character by making it an array of characters. (The concept of arrays is thoroughly discussed in Chapter 8; however, a brief introduction to them is necessary for you to understand strings in C.) For instance, the following declaration

```
char a[10];
```

would allocate space in memory to hold a string of up to ten characters. The name "Oliver" could be put into this string, as could the name "Atari." When we show the value of a string, it is shown surrounded by double quotes (" "). If you were to look into the actual memory location occupied by the variable **a**, you would see the following when "Oliver" is in it:

```
'O''l''i''v''e''r''\0'
```

and this when "Atari" is in it:

```
'A''t''a''r''i''\0'
```

As you might have guessed, each letter in the names occupies one byte of memory, but then it is followed by a '\0', or null character. Remember, part of the definition of a string is that there must be some termination signal, and that is what the null character represents. Although '\0' appears to be two characters, it is really just one character value that requires two characters to show its meaning. Later we will be seeing other escape or control characters like the null character, but for now you only need to understand that '\0' tells C that this is the end of the string.

We can access individual characters within a string by indexing the variable like this:

```
a[4]
```

which would have the value 'e' and 'i' when the string was "Oliver" and "Atari" respectively. You may have thought the value of a[4] would be 'v' and 'r', but the C language uses zero as the beginning of its array indices. So, instead of being able to look at any of the characters a[1] through a[10], the proper indices would be a[0] through a[9].

What to Declare?

In general, C provides the ability to mix and match variables of different types via loose type checking. For this reason, it is possible to assign almost any type of variable to another variable of a completely different type. This functionality can be good in the sense that it allows for a great deal of flexibility in coding and may alleviate conversion problems for the programmer. However, it can also cause serious problems if not used properly. For this reason, it is recommended that as a beginning C programmer, you should not attempt to take advantage of this advanced feature. Wait until you have written several lengthy programs and feel completely comfortable with C before mixing and matching!

Before moving on, we want to emphasize that when providing variable identifiers, we strongly recommend using identifiers that are descriptive or meaningful to the variable defined. As stated above, identifiers can be several characters long, although only the first eight are significant to the compiler. While there is nothing technically wrong with an identifier defined by a single character, readability should be a major consideration for proper program documentation. For example, a variable that represents someone's grade in a class is better suited by an identifier that is descriptive of its use rather than the letter *x*. For example, we can use

```
float grade;
```

Assigning Variables

In our box example, we state that a variable inside the box, or memory location, can contain any value. To do this, we need a method for inputting

values into a memory location. We can assign values to a variable by using the assignment statement. The assignment statement has the following general syntax:

```
variable = expression
```

where the variable on the left is assigned the value of the expression on the right. The symbol = is an assignment operator. If, for example, the variable represents a legal numeric variable identifier, an assignment statement might be:

```
age = 29;
```

where the value 29 is assigned to the memory location defined by the variable age. It is important to note that using the assignment operator here is not the same as using the equal sign used in basic algebra. In algebra, two values may be considered equal in value. C's assignment operator simply means a variable is "assigned to" or "is given or becomes the value of" the expression. As a final note on C variable assignment, the assignment may take place at the point of the declaration, like this:

```
int a = 4;
```

which results in space being allocated for **a** and that space being initialized to 4. This form of variable initialization comes in handy quite often. But keep in mind that because the value is assigned in the declaration part, it may be easy to overlook when you are reading through the code and could cause severe headaches when you are trying to track down a problem.

Classes of Storage

As the earlier pseudo–C code listing showed, C programs consist of a main routine and perhaps several other routines or functions. Up to this point, all the variables we have discussed are called automatic, or auto, variables. This is because the variable is automatically assigned a space in memory for that routine. In fact, we could have preceded all of our variable declarations with the reserved word **auto** like this:

```
auto int x;
```

but the default storage class is automatic, so it is not necessary. In general, storage space for an auto variable is allocated whenever the function in which it resides is invoked, and when that function has completed execution, the storage space is relinquished for other program usage. For this reason, if a function contains an auto variable, you can never assume that that variable still contains the same value that was in it the last time the function was invoked.

Alternatively, a variable declared to have the *static* storage class will always still have the most recent value assigned to it even if it was assigned several invocations ago. A static variable is declared like this:

```
static int x;
```

A third storage class of variables is the register type. By preceding the variable declaration with the reserved word **register**, like this:

```
register int x;
```

on some systems and with some compilers one of the microprocessor's (the "brains" of the system) registers will be used to hold the variable's value. This sort of declaration may help to speed your program up if you use it with a variable that is used quite intensely. Registers on the Atari ST are four bytes long, so keep in mind that only variables that have a data type of four bytes or less may be declared as register variables.

The last storage class of variables is the *extern* class, which simply informs the compiler that this variable name is used externally to this routine, and both references to it are for the same space in memory. For example, with the following declarations

```
int x;
char y;
main()
{
  extern int x;
  extern char y;
```

you are informing the compiler that you know *x* and *y* were declared earlier outside of *main( )* and that all references to both of them are for the memory space allocated by the external declarations. In this situation, the extern dec-

larations are not necessary because both sets of declarations are in the same file. But when you later progress to writing lengthy programs that are split up into multiple source code files, the extern declaration can come in quite handy.

Expressing Ourselves

We have introduced the concept of an expression during our discussion of the assignment statement. An *expression* is a sequence made up of variables, mathematical operators, and constants (via the **#define** directive). The following are valid expressions:

```
6 + 2
a + b - 3
pay - 3.25
sum + 1
```

As you recall, we use the assignment operator to assign the value of an expression to a memory location defined by an identifier. The operators used in an expression must be valid for the type of data they operate on. The most commonly used expressions are arithmetic. C is capable of performing most of the mathematical operations that you are familiar with:

+	addition
−	subtraction
*	multiplication
/	division
%	modulus or modulo division (remainder from integer division)

In C, there also exists a unary minus (−) operator that may be used to change the sign of the operand. You should recognize the operation of +, −, *, and /; however, the operator % (modulo division) may be new to you. The modulo operator is used to calculate the remainder of integer division. For example:

6 % 5	returns a value of 1
8 % 4	returns a value of 0

9 % 6 returns a value of 3
9 % 2 returns a value of 1

The modulo operator has no meaning on real values and, as such, may not be used with them.

Mathematical Order of Operation

Mathematical operations are performed in a particular order. For example, multiplication and division are performed before addition or subtraction. To illustrate, what do you think the result of this operation will be:

```
3 + 5 * 6
```

If you read this example from left to right, you might interpret the result as 48 instead of 33. The multiplication (5 * 6) is done before the addition (3 + 5). This is a result of the order of operator precedence. Operators with the highest precedence are performed before operators with the lowest precedence. The unary (−) operator has the highest precedence of the mathematical operators. After it, the high precedence operators are *, /, and %. Operators with low precedence are + and −.

If an operation consists of operators of the same precedence, then the calculation may be performed from left to right. In the example below, the left operation (addition) may be calculated first, followed by the second operation (subtraction).

```
6 + 9 - 2
```

As you probably determined, the result of this expression is 13.

Parentheses are used to dictate the order of operations—the same way they are used in basic mathematics. The operations contained within parentheses have a higher order of precedence than multiplication or division. For example, consider this assignment statement:

```
i = (4 + 8)/2;
```

The result is that the variable *i* is assigned a value of 6. The operation contained within the parentheses is performed before the division (/) operation.

Operations contained within a set of parentheses must follow the normal order of precedence. Can you determine the correct value of this statement:

```
rate = (6 * 2 + 18) - (8 - 16/4);
```

The correct value is 26.

In summary, arithmetic expressions consist of a sequence of variables, operators, and/or constants. To eliminate any ambiguity in an arithmetic expression, a mathematical order of operations, or operator precedence, must be followed.

Simple C Arithmetic

We have covered several fundamental concepts in this chapter, and so it is appropriate that we should review them in a couple of simple program examples. Consider the task of calculating your gas mileage—a simple problem that C on your Atari ST can easily calculate. How many miles do you travel to work or school? How many gallons of gas do you use? You know that mileage can be defined as the number of miles traveled divided by the number of gallons of gas used, right? If you answered these questions, you have defined the programming problem. If you determine that you traveled 200 miles last week on 8 gallons of gas, you can write a program that looks like this:

```
/*This program calculates your gas mileage.*/

#include <stdio.h>

float miles, gas, mileage;

main()
{
miles = 200;
gas = 8;
mileage = miles/gas;
printf("%f", mileage);
getchar();
}
```

When you run this program (after compiling and linking it), the result

is displayed in the upper left-hand corner of the display as 25.000000. Let's quickly review this program step by step. We won't explain every program in such excruciating detail, but it is important that you master these beginning concepts early.

The first line is a comment that briefly describes the program's purpose. The second line is our old friend, the **#include** directive, which tells the compiler to pull in the **stdio.h** header file. Next, our three float-type variables are declared. Notice that all three variables are declared on the same line with only one reference to the float type. This is quite acceptable in C and is highly recommended because it decreases the number of keystrokes necessary to enter programs in the editor.

After our variables, we being the **main( )** routine, which is surrounded by the beginning and ending bracket pair ({ and }). The program statements within the program block are the executable portion of the program. The first three lines, following the begin bracket ({), are all assignment statements. Each line is concluded with a semicolon to separate the individual program statements. The next statement is the **printf** statement. This statement will be covered in detail in the next chapter. All you need to know now is that it is responsible for displaying the result of our mileage calculation. In short, the information contained within the parentheses of the **printf** statement tells the compiler that we wish to display the value of the variable **mileage** in a "float" (%f) format. Finally, the last statement is a call to the **getchar( )** routine, which we used earlier to allow the screen to stay unchanged until you press the RETURN key.

The next example further illustrates the use of mathematical variables and assignment statements. Study the following program, and determine if you can understand each of the program's components. If you can't, you should review the appropriate section.

/*This program calculates a student's grade point average*/

```
#include <stdio.h>

float test1, test2, test3, test4;
float average;
char name[] = "Joe";

main()
{
test1 = 85;          /*the following are assignments*/
test2 = 90;
test3 = 78;
```

```
test4 = 79;
average = (test1 + test2 + test3 + test4)/4;
printf("What is the student's name and test average?\n");
printf("%s\n", name);
printf("%f", average);
getchar( );
}
```

When you execute this program, your screen should read:

```
What is the student's name and test average?
Joe
83.000000.
```

Besides a couple of more **printf** variations, the only new item presented in this program is the ability to declare a string without explicitly specifying its length in the declaration. The compiler will determine the length for you based on the length of the string we have initialized it to (in this case, "name" occupies 4 bytes; don't forget the trailing '\0' termination character). Finally, let's informally discuss how your programs should look in appearance, a concept called program formatting.

Program Formatting

Program formatting is a very important concept in C programming. In general terms, program formatting is the position on your screen or line that you begin a statement, insert a space, indent, etc. The reason for program formatting is more than a compulsive desire for neatness. Its purpose is to make your programs more readable and thus reduce the amount of time needed for debugging or comprehension of a program by someone other than the programmer who created it.

Indentation will become a major factor in proper program formatting as we get into more complex source code. It is recommended that every time an indentation is necessary, you should indent a fixed number of spaces, say, 3 or 4. This way, your programs will take on a much more professional appearance and will lend themselves to easy tracing by other programmers.

We have covered a lot of ground, so we recommend that you take a 15-minute break, absorb what you have learned thus far, and then continue on

to Chapter 3. If you don't completely understand all the concepts presented in this chapter, go back and reread the appropriate sections.

Glossary for Review

syntax—the rules of a language that must be followed to communicate meanings properly

bit—binary digit (one or zero)

byte—group of 8 bits

Quiz

1. What is wrong with this statement?

   #define top 10

2. What are the two ways of including a separate source file in your code?
3. Which data type occupies more memory—int or unsigned int?
4. What does '\0' mean in a string?

Chapter 3

C Statements

In this chapter, you will learn:

- How to communicate with the user; the Input/Output statements.
- What the **printf** and **putchar** routines are, and how to use them.
- What the **scanf** and **getchar** routines are, and how to use them.
- How to write programs that make decisions.

This chapter takes a deeper look into C. You have learned many of the fundamentals of program construction in the previous chapters. Most of the material you are about to read discusses various program statements that enable you to write more complex programs. As you recall, program statements are the executable statements that appear within a program block, or { and } pair. We will explore fundamental control structures that give the computer the ability to make decisions. In addition, we will study interactive statements between the computer and the user. First, however, we need to examine how information is displayed, or output, to your screen. Let's start by taking a closer look at these output routines.

The printf Function

You were briefly introduced to the **printf** function in the previous chapters. In this section, we cover the **printf** function and related output functions in more detail. These routines are very fundamental to programming in C. In fact, these are the functions that C uses to communicate the output processed by your programs so that you can readily understand its meaning. The general syntax for the **printf** function is:

```
printf("cont_strng", arg1, arg2, . . ., argn);
```

where **cont_strng** is the control string that specifies how you want the output formatted, and **arg1**, **arg2**, etc. are the actual arguments that are to be displayed. We have already seen examples of how **printf** can be used to print out a literal string like this:

```
printf("Welcome to C");
```

In this instance, the control string consists of a series of characters that we want to put directly on the screen. That is, there are no argument parameters used in this example. However, as our syntax notes, **printf** can be used to display both literal strings and other values as passed in the arguments. For example, if we want to display the value of the int variable **goals**, we would write it like this:

```
printf("%d", goals);
```

In the control string, the % character signifies that the next character will be a conversion character specifying the format of the accompanying argument. In this case, the conversion character is a *d*, meaning decimal format. The nine conversion characters and their translations are:

Conv. Char.	*Meaning*
c	print a single character
d	print a decimal value
e	real (float) number; display in exponential notation (e.g., 4.56 E − 5)
f	real (float) number; display in decimal form (e.g., 45600000.00)
g	use the shorter of e or f above
o	display in unsigned octal form (base 8)
s	print a string
x	display in unsigned hexadecimal form (base 16)

In addition to displaying just the arguments in the parameter list, we can mix literals in with the conversion characters to display statements with variables like this:

```
printf("Mike is %d years old.", age);
```

where the *%d* references the int variable *age* and would produce the following output (if age is set to 25):

```
Mike is 25 years old.
```

You could also have set up a string variable like this:

```
char name[] = "Mike";
```

and then written the **printf** like this:

```
printf("%s is %d years old,", name, age);
```

and you would have achieved the same results.

All of your output has been on one line so far, but, of course, this need not always be the case. The control character \n may be used in **printf** statements to advance the output position the next line down on the screen. For example:

```
printf("Mike is %d years old\n.", age);
printf("He is almost as old as %s.", name2);
```

will display the following:

```
Mike is 25 years old.
He is almost as old as Mary.
```

if, of course, **age** is equal to 25 and **name2** is equal to Mary. These new line characters may be interspersed in your control string as you see fit to format your output properly.

Speaking of output formatting, you may have noticed that **printf** will always display a float variable with 6 decimal positions even though you may only be interested in a couple of them. Consider the following declarations and C statements:

```
float miles = 300;
float gallons = 12;
float mpg;

mpg = miles/gallons;
printf("You got %5.2f miles per gallon\n", mpg);
printf("You got %5.2f miles per gallon\n", miles/gallons);
```

The output produced by the two **printf** statements will be:

```
You got 25.00 miles per gallon
You got 25.00 miles per gallon
```

because the **%5.2f** control segment specifies that you desire a float value placed in a field that is five characters wide, and two of those characters, at most, are to be for the fractional portion of the number.

You now know the basic concepts behind the **printf** function. Try a few short programs of your own where you print out combinations of floats, ints, and strings so that you thoroughly understand the principles.

The putchar Function

In addition to the **printf** function, **putchar** is available to display one character at a time. The syntax of **putchar** is:

```
putchar(ch);
```

where "ch" is a char value. So the following declarations:

```
char a = 'a';
char b = 'b';
char c = 'c';
```

and **putchar** statements:

```
putchar(a);
putchar(b);
putchar(c);
```

will display the characters *abc* in the upper left-hand corner of your screen.

Now that we have an understanding of the fundamental output functions available in C, let's take a look at the functions available to receive input from the user.

The scanf Function

The **scanf** function is quite similar to **printf**, and its syntax is:

```
scanf("cont_strng", arg1, arg2, . . ., argn);
```

where the control string characters have the same meaning as those in **printf** except that there are no %u, %e, or %g, but there is a %h, which may be used to read in a short int. So, if we want to elicit input from the user, we could say:

```
printf("What is your first name?\n");
scanf("%s", fname);
```

Whatever you enter in at this prompt would then be put into the *fname* variable location. We could then print it back out like this:

```
printf("Your name is %s.", fname);
```

Take a look at this interactive example:

```
/*This program asks for your name and age and displays them.*/

#include <stdio.h>

char fname[10];
int age;

main()
{
printf("What is your first name?\n");
scanf("%s", fname);
printf("How old are you?\n");
scanf("%d",&age);
printf("\n\nYour name is %s and you are %d years old . . .", fname,
age);
getchar();
}
```

This program first asks you to enter your first name, and then it asks you for your age. The **scanf** call for inputting your age is a little bit different than the one used for your name, though. There is an ampersand (&) before the age parameter. This ampersand means that scanf is not using the actual **age** variable but rather a pointer to it. It is not really necessary for you to know the difference between the two here, since pointers will be discussed later. But you do need to know that for all arguments other than strings, you need to precede the variable name with an ampersand in the parameter list. Other than this difference, the **scanf** function is very easy to understand. So now let's take a look at another routine that allows you as a programmer to accept input one character at a time.

The getchar Function

You have already seen getchar used in some of the earlier programs in this book as a method of making the program stop so that you can look at the screen before the GEM desktop is redisplayed. The syntax of getchar is:

```
ch = getchar();
```

Notice that ch, which is taken to be an int variable, is not passed as a parameter to getchar, rather it is the value returned by the function. We will deal with functions and their methods of returning values in Chapter 5, but for now, when you see a variable on the left side of a statement and a function call on the right side, you may assume that the variable takes on the value returned by the function. So, we could accept characters and display them back out to the screen by having getchar and putchar right after each other like this:

```
ch = getchar();
putchar(ch);
ch = getchar();
putchar(ch);
ch = getchar();
putchar(ch);          /* and so on . . . */
```

You have now been exposed to enough input/output routines to allow you to write even fairly sophisticated programs. But, in order for you to make your ST really work for you, you need to learn about how you can instruct it to make decisions.

Statements of Choice: The Conditionals

A fundamental feature of any computer is its ability to make decisions. The computer makes its decisions based on certain conditions or tests. If a particular condition is met, the flow of control within a program is modified, and a different set of instructions is performed. Changing the flow of control of a program is essential to writing powerful programs. The programs de-

picted in this book thus far simply execute each statement sequentially. Such step-by-step execution limits the programmer's ability to develop complex and useful programs. In the next section, we will learn how to change the flow of sequential instructions by looking at some fundamental control statements: the "if . . . else" compound statement and the switch statement.

Decisions, Decisions . . . The Thinking ST

For practical applications, the computer must have the ability to make decisions based on a conditional. What do we mean by a conditional? If, for example, you ask a question, and the response can only be true or false, such a response is an assertion to the circumstance of the question. The statement in C that creates the conditional is called the "if" statement.

The "if" statement makes a decision by performing a decision test. The test has a true or false, yes or no, or precisely one or zero result based on the conditional expression. We say "one or zero" because the only way any computer can think is in terms of ones and zeros as bits. The syntax of the "if" statement is written as a compound statement, with the associated statement and the optional "else" statement. The structure of the simple "if" statement provides two related choices. The first option is:

```
if (expression is nonzero)
  {
  statement1;
  statement2;
  statement3;
  }
statement4;
```

This first option states that if the expression is nonzero, statement1 is executed, followed by statement2, etc., until the end of the block. Then statement4 would be executed. The conditional block is enclosed within the familiar beginning and ending brackets ({ and }). If the expression is zero, the conditional block is bypassed, and the next statement to be executed is statement4.

A second structure of the "if" statement can be written as:

```
if (expression is nonzero)
  statement1;
```

```
else
  statement2;
statement3;
```

The choice above adds a second option, the "else" portion, to our compound statement. The addition of "else" provides the possibility of one more statement being executed if the expression evaluates to zero. Thus, if the expression is nonzero (or true), statement1 is executed, statement2 is skipped over, and statement3 is then executed. If the expression is zero (or false); statement1 is skipped over, statement2 is executed, and then statement3 is executed.

As we have said, when a conditional expression is executed, a relationship is compared to determine if a condition is true or nonzero. If the condition is true, then the next statement or block of statements is executed. The condition itself is specified by relational clauses or operators. The relational operators used in C are:

==	(equal)
!=	(not equal)
<	(less than)
>	(greater than)
<=	(less than or equal to)
>=	(greater than or equal to)

This simple example illustrates the conditional "if" statement:

```
/*This program shows how to use the "if" statement.*/

#include <stdio.h>

int num1, num2;

main()
{
printf("Enter a number:");
scanf("%d",&num1);
printf("Enter another number:");
scanf("%d",&num2);
if (num1 == num2)
  printf("Both numbers are equal!\n");
```

```
else
  printf("Both numbers are not equal!\n");
printf("Isn't this fun?!");
getchar( );
}
```

Let's study this program. If you obey the computer and enter the numbers 4 and 2, then the second **printf** statement is executed and displays:

```
Both numbers are not equal!
Isn't this fun?!
```

Your curiosity gets the better of you, however. You decide to test the computer to see if it's awake. You run the program a second time, but now you enter the same number twice. This time your ST responds with:

```
Both numbers are equal!
Isn't this fun?!
```

Why? Our second test resulted in a condition that was true. Under a true condition, the first **printf** statement is executed. Our first test resulted in a false condition; therefore, the first **printf** statement is skipped over, while control of the program passes to the next available statement.

A Couple of Common Errors with "if" Statements

Beginning programmers in C often make a common mistake when using the "if" statement. The mistake is a logical one where the output received is an unexpected value. Specifically, we are referring to the result of the conditional when the condition is false and more than one statement follows the check.

As you recall, when a condition is false, program flow skips the first statement and then continues with the first statement after the affirmative block. Execution, however, continues from that point, and the next statement is executed in successive order. This truth may not provide the results desired. To illustrate, let's examine a portion from our previous example:

```
if (num1 == num2)
  printf("Both numbers are equal!\n");
```

```
else
  printf("Both numbers are not equal!\n");
printf("Isn't this fun?!");
```

When the value of **num1** is equal to **num2**, the condition is true and the program flows to the first **printf** statement and then to the third **printf** statement. However, suppose you don't want the third **printf** statement to be executed when the condition is true. At present, this situation is not possible because the program flow will simply continue from the first to the third in a true condition. To do this, we can include all the alternatives in a single block of statements. The statements within the compound statement ({ and }) are executed in sequence. Consider this modification to our program:

/*This is a modification to our program which shows how to use the "if" statement.*/

```
#include <stdio.h>

int num1, num2;

main()
{
printf("Enter a number:");
scanf("%d",&num1);
printf("Enter another number:");
scanf("%d",&num2);
if (num1 == num2)
  printf("Both numbers are equal!\n");
else
  {
  printf("Both numbers are not equal!\n");
  printf("Isn't this fun?!");
  }
getchar();
}
```

In this situation, the message "Isn't this fun?!" is displayed only whenever two different numbers are entered. Because it is within the block of the "else" portion, as designated by the begin and end brackets ({ and }), it will never be executed when two identical numbers are entered.

Another error with "if" statements that is usually encountered by programmers familiar with other languages (especially PASCAL) is the use of

a semicolon at the end of the statement before the "else" clause. As all C programmers know, the semicolon is used at the end of all executable statements, but sometimes it is difficult to remember to use it even before an "else" if you are used to leaving it off!

Also, always indent statements within a compound block (as shown above) so that it is easy to trace which statements are meant to go with which block. It may not appear to be very important now, but when you start writing more sophisticated code, you'll be glad you did it.

Nesting Your "if" Statements

The statements that follow the true or false clauses may be of any kind. The statements can be **printf** statements, assignment statements, compound statements ({ and }), or even another "if" statement. When an "if" statement appears within another "if" statement, the statement is called a compound "if" statement. The technique of using multiple "if" statements within the same program structure is called **nesting** the "if" statement. There is no steadfast restriction to the number of "if" statements that may appear nested within each other; this generally changes from compiler to compiler. However, such a complicated structure can become very difficult to follow logically. A sample syntax of nested "if" statements might be:

```
{
if (condition1)
  if (condition2)
    statement1;
  else
    statement2;
  else
    statement3;
}
```

Notice the successive indentation of each "if" statement. Again, this formatting makes programs much easier to understand.

Why use a nested "if" statement? The answer is simple. Suppose you want a program to make a decision and then to perform one of two exclusive actions. In this case, an "if" statement will satisfy your needs. Suppose, however, that you want the computer to provide three alternative actions instead of just two. To remedy this problem, we imbed or nest an additional decision test. To illustrate, let's study the following example of calculating an employee's pay:

```
/*This program calculates pay including overtime*/

    #include <stdio.h>

    #define RATE 4.00
    int hrs;
    float pay;

    main()
    {
    printf("How many hours worked?");
    scanf("%d",&hrs);
    if (hrs > 40)
      if (hrs > 45)
        {
        pay = (RATE * 40.0) + (2.0 * (RATE * (hrs - 40)));
        printf("You earned double OT! Your pay is $%10.2f", pay);
        }
      else
        {
        pay = (RATE * 40.0) + (1.5 * (RATE * (hrs - 40)));
        printf("You earned OT! Your pay is $%10.2f), pay);
        }
    else
      {
      pay = RATE * hrs;
      printf("Your pay this period is $%10.2f", pay);
      }
    getchar();
    }
```

When the program is executed, the user is requested to input the number of hours worked. If a value of 40 hours or less is entered, a straight-time pay rate ($4.00 per hour) is provided. If a value greater than 40 hours but not exceeding 45 is entered, the employee earns overtime pay. A value of greater than 45 hours results in double overtime pay. For example, if you enter 40, the screen displays:

```
Your pay this period is $ 160.00
```

If you respond by entering 45 instead, the output is:

```
You earned OT! Your pay is $ 190.00
```

Lastly, if you enter 50 instead, then the output is:

```
You earned double OT! Your pay is $ 220.00
```

In our sample program, the nested "if" statement provides three courses of action. Only one statement or statement block may follow the affirmative, or true clause, and appear before each "else."

Logical Operators

We have not discussed logical operators. A conditional or logical expression is offered a greater amount of flexibility with the logical operators || (or), && (and) and ! (not, or negation). These operators give you the ability to write compound expressions. For example, suppose you input a value that you want tested to be between the value of 1 and 10. We can write:

```
if (num > 1) && (num < 10)
```

where the expression is only true if both assertions are true. Notice that both assertions are enclosed within a set of parentheses and separated by the "and" operator &&. The individual should be enclosed within parentheses to assure your intent is understood.

The remaining two operators function in a similar fashion. The logical operator || (or) states that only one (or both) of the individual assertions must be true for the entire expression to be true. The logical operator! (not) provides the exact opposite of a truth value and is called the logical negation (i.e., not true is false). The result of the three logical operators can be summarized in what is called a truth table as follows:

A	*&&*	*B*	*Result*	*A||B*	*Result*	*!A*	*Result*
T		T	T	T T	T	T	F
T		F	F	T F	T	F	T
F		T	F	F T	T		
F		F	F	F F	F		

The ?: (Conditional) Operator

Through the simple "if" statement we have the ability to write a portion of code like this:

```
if (hrs < 20)
  pay = hrs * 2.00;
else
  pay = hrs * 4.00;
```

where if the condition (hrs < 20) is true, then the pay will be based on a $2 per hour rate; otherwise pay will be based on a $4 per hour rate. C provides an operator that may be used to abbreviate this sort of "if . . . else" condition and it looks like this:

```
pay = (hrs < 20) ? (hrs * 2.00): (hrs * 4.00);
```

If the condition (hrs < 20) is true, the first expression to the right of the question mark (hrs * 2.00) will be evaluated with the result assigned to the variable on the left side of the equal sign (pay); otherwise, the expression after the colon (hrs * 4.00) will be evaluated with the result assigned to the variable. This operator may take some time to get used to, but it can cut out a few extra lines from your source file.

The Switch Alternative

As we have discussed, the opportunity exists for you to write a segment of code that checks a variable for a specific value and, based on that value, then executes certain statements. We have seen this in "if" statements like this:

```
if (num == 5)
  printf("The number is five");
else
  printf("The number is not five");
```

With just two alternatives, this sort of statement is acceptable. However, what if we had to display a unique message if the number was any integer from one to five? The "if" statement would look like this:

```
if (num == 1)
  printf("The number is one");
else if (num == 2)
  printf("The number is two");
else if (num == 3)
  printf("The number is three");
```

```
else if (num == 4)
  printf("The number is four");
else if (num == 5)
  printf("The number is five");
else
  printf("The number is not within one to five");
```

As you can see, it is allowable to have progressive checks in an "if" statement via the "else . . . if" option. This form of the "if" statement checks each condition and, once it finds a true one, it executes the statement block associated with that condition and then skips over the rest of the "if" statement.

This form of the "if" statement is indeed quite useful, but C provides another statement, the switch, that is much more readable than multiple else . . . if's. The same lengthy "if" statement looks like this as a switch statement:

```
switch (num)
  {
  case 1: printf("The number is one");
          break;
  case 2: printf("The number is two");
          break;
  case 3: printf("The number is three");
          break;
  case 4: printf("The number is four");
          break;
  case 5: printf("The number is five");
          break;
  default: printf("The number is not within one to five");
  }
```

As you might have guessed, the expression **(num)** is evaluated, and its value is compared against each of the values specified by the reserved word **case**. When a match is found, that case's statement block is executed. However, at this point, the switch statement is not intelligent enough just to jump over the remaining checks. Rather, the break statement must be used to inform C that you do not wish to execute the very next statement. This is also why a begin/end bracket set ({}) is not necessary with any of the case statement blocks. Finally, if the expression does not equal any of the case values, the default block is executed. Note that there is no need for a break statement at the end of the default block because we are already at the end of the switch statement!

You have just completed another large portion of C programming basics. In Chapter 4 we cover more C statements that provide additional control over the flow of your programs—the looping statements. It's time now to review what we have learned so far. Again, if you have trouble with a particular section, review it first before continuing on to Chapter 4.

Quiz

1. What's wrong with this statement?

```
scanf("%d", num);
```

2. How are *getchar( )* and *putchar( )* related?

3. What is printed in this "if" statement?

```
if (num)
  printf("The number is non-zero");
else
  printf("The number is zero");
```

Chapter 4

More Statements: The Looping Structures

In this chapter, you will learn:

- How to use the "while" statement to construct a programming loop.
- How to use the ++, --, +=, -=, *=, /=, and %= operators.
- What a counting and summing loop is, and how to use one.
- How to use the "for" statement in a looping structure.
- How to use the "do-while" statement to construct a loop.
- What a nested loop is, and how to use one.

In this chapter we are going to continue our discussion of C statements. In particular, we will examine a few additional control structures that allow you to write programs that repeat or loop. Looping programs, again, allow you to alter the flow of a program or its physical order of execution, sequentially, from the first program statement to the last. C offers us several looping constructs to choose from: the "while" statement, "for" statement, and "do-while" statement. Let's begin by looking at one of our favorites, the "while" statement.

Programs That Repeat

Thus far, we have discussed programs that are executed one line at a time. To execute a program; you double-click on its icon on the GEM desktop. If you wish to execute the program again, you must double-click on it a second time. One command in C allows you to execute a program, or a block within a program, repeatedly without double-clicking over and over again. To do this, we can write a program that repeats itself over and over until a certain condition is true. Such a conditional control is called the "while" statement.

The "while" Statement

Similar to the "if" statement, a "while" statement tests a condition to determine which of two courses of action to take. However, unlike the "if" statement, after the body of the "while" statement executes, the program flow "loops" back to the start of the "while" statement to test the condition again. As long as the tested condition is true, or nonzero, the first action is repeated endlessly until the condition tested is false, or zero. A sample syntax of the "while" loop can be written as:

```
while (expression)
  statement1;
statement2;
```

where the reserved word "while" is followed by a logical expression, and statement1 may represent one statement or a block of statements enclosed within the { and } brackets. The expression is, of course, a true or false

condition. If the expression is a true test, the program will execute the statement (or statement block) once. After this execution, the program loops back to test the expression again. This looping construct continues executing the body of the loop until the expression results in a false, or zero, value. A false test tells the computer to skip statement1 (or the statement block) and execute statement2. At this point, the looping "while" statement is completed, and the program will execute the next available statement.

A special example of the "while" loop appears in the following program. We provide this special program to illustrate a point. In this example, you are requested to enter a number. Any response other than 13 will make the condition true and continue to display the statement following the "while" check endlessly. Such a loop is called an **infinite** loop. To stop an infinite loop on your ST, hold down the ⟨Control⟩ key and press the C key. Pressing the ⟨Control⟩ key while pressing another key is commonly written as CONTROL/key, or, in our situation, CONTROL/C. This key combination has the effect of telling the computer to stop the program in progress. After you press CONTROL/C, the GEM desktop will be redisplayed, and you can run the program again or run something else. If the number entered is 13, the loop is avoided.

/*First "while" loop program*/

```
#include ⟨stdio.h⟩

#define MY_NUM 13

int num;

main()
{
printf("Please enter a number:");
scanf("%d", &num);
while (num != 13)
  printf("You entered the number %d.\n", num); /* infinite loop */
printf("You entered the number 13!!!");
getchar();
}
```

Try running the program above. A response of 13 causes the second **printf** to be executed, and the loop is avoided. To see the effect of the infinite loop, enter a number other than 13. The screen fills up so fast with the same line it is almost hard to see each new one being displayed.

The example above illustrates poor programming technique. Infinite loops should be avoided. A more practical use of the "while" loop is a structure where the loop is executed a specific number of times. This type of technique is discussed next.

Counting Your Loops

To control the number of times a loop executes, we can insert a counter that accumulates the number of iterations that occur. A counter is actually a variable that is assigned a value which increments each time the loop is executed. When the expression of a "while" statement reaches some predefined value, the condition results in a false value, and the flow of control moves out of the program loop. To illustrate, try this program:

/*This program calculates products in a loop.*/

```
#include <stdio.h>

int count, loops_left;
int num1, num2;

main()
{
count = 0;
while (count < 5)
  {
  printf("Enter a number:");
  scanf("%d",&num1);
  printf("Enter another number:");
  scanf("%d",&num2);
  printf("The product of this set is %d\n", (num1 * num2));
  count = count + 1;
  loops_left = 5 - count;
  printf("This loop has executed %d time(s).\n", count);
        printf("This program will loop %d more time(s).\n\n",
        loops_left);
  }
printf("\n\nTHIS PROGRAM HAS ENDED!!!");
getchar();
}
```

This program illustrates two important points. First, notice the construction of the "while" statement. The first course of action following the "while" line is a compound statement ({ and } pair). As we have said before, the compound statement may contain any number of statements and still be considered a single block by the construct of the "while" loop. In other words, referring back to the syntax of the "while" loop, the compound statement represents statement1.

The second point of this example is the construction of the loop counter variable. We have appropriately called ours **count**. The counter is first initialized to zero outside the loop. Within the loop, the counter is incremented by one each time the "while" loop executes. When the condition of the "while" statement is false, program flow exits from the loop and informs the user that the program has ended. The loop itself executes exactly five times. How do we know this? The number of times the loop executes is determined by the loop control variable, in this case the variable **count**. We compare the value assigned to the variable when it was initialized (count = 0;) to the value of the condition test when the test is false (count < 5). We can set the count equal to 1 and then set the condition test as (count <= 5). Either way, the loop executes five times. Be careful when setting the number of controlled loops. If the counter above was initially set to zero and the decision test was set as (count <= 5), the loop would execute six times and not five!

Loops That Sum

We can also write a program that accumulates a result within a loop. Such a loop is used to sum a list of values and is called an **accumulator**, or **summing**, loop. This type of loop looks just like the counter just discussed; in fact, the difference between the two is almost negligible. The point to be made is that a summing loop doesn't need to be dependent upon the value of the counting variable within a "while" loop. The two loops, a counter and an accumulator, may coexist within the same "while" statement. For example:

```
/* This program sums the five numbers you enter. */

    #include <stdio.h>

    int num, count, sum;
```

```
main()
{
printf("This program will sum a list of number.\n");
printf("You will be prompted to enter 5 numbers and you\n");
printf("should press <RETURN> after each entry.\n");
count = 0;
sum = 0;
while (count < 5)
  {
  printf("Enter a number:");
  scanf("%d",&num);
  sum = sum + num;
  printf("The sum of your numbers so far is %d\n", sum);
  count = count + 1;
  printf("We have made %d pases through the loop.\n\n", count);
  }
printf("The program is over and the sum of your numbers is %d.",
sum);
getchar();
}
```

As you can see, the variables **count** and **sum** are used to see how many times we have been through the loop and the total of the numbers entered so far, respectively. A sample run of the program might look like this:

```
This program will sum a list of numbers:
You will be prompted to enter 5 numbers and you should press
<RETURN> after each entry.
Enter a number: 1
The sum of your numbers so far is 1
We have made 1 passes through the loop.

Enter a number: 2
The sum of your numbers so far is 3
We have made 2 passes through the loop.

Enter a number: 3
The sum of your numbers so far is 6
We have made 3 passes through the loop.

Enter a number: 4
The sum of your numbers so far is 10
We have made 4 passes through the loop.
```

```
Enter a number: 5
The sum of your numbers so far is 15
We have made 5 passes through the loop.

The program is over and the sum of your numbers is 15.
```

Every time you enter a number, the sum so far is reported, as is the number of times we have gone through the loop. After five passes, a message is displayed showing you that the loop is finished, and your final total is shown. Before we look at another way of looping in your programs, let's look at a few abbreviations C provides for some simple mathematical operations that come in quite handy, especially when you are working with loops.

More Mathematical Operators

In the example above, the statement we used to increment our count variable was:

```
count = count + 1;
```

which is rather verbose. Fortunately C provides this shorthand notation for incrementing variables by one:

```
var++;     OR     ++var;
```

where var++ increments the variable after its value has been used (post-incrementing), and ++var increments it before its value has been used (pre-incrementing). The difference between the two makes no difference to us in our program, since we can simply replace the line:

```
count = count + 1;
```

with:

```
count++;
```

or:

```
++count;
```

Either way, the variable **count** will be incremented every time either of these statements is executed. To see exactly how the two formats differ, try this program:

```
/*This program shows the difference between ++var/var++*/

#include <stdio.h>

int num1;
main()
{
num1 = 1;
printf("With ++num1, we get %d,\n", ++num1);
printf("with num1++, we get %d,\n", num1++);
printf("and the final value of num1 is %d.", num1);
getchar();
}
```

Although you should not be surprised to discover that the value printed in the first **printf** is 2, you may be puzzled to see that the second **printf** also displays a value of 2. As we said, num1++ does not increment the variable **num1** until after it's current value has been used in that statement. This is why the third **printf** shows the value of **num1** to be 3, which is what we expect it to be after incrementing it twice.

C also provides the same type of incremental operators for subtraction, namely:

```
var--;    AND    --var;
```

where the same rules of post-decrementing and pre-decrementing apply. In addition to these mathematical shortcuts, C also provides the following abbreviations, which are shown with their general equivalents:

General Form	*C Abbreviation*
num1 = num1 + num2;	num1 += num2;
num1 = num1 - num2;	num1 -= num2;
num1 = num1 * num2;	num1 *= num2;
num1 = num1 / num2;	num1 /= num2;
num1 = num1 % num2;	num1 %= num2;

So, if for some reason we would have wished to increment our variable by, say 3, every time through a "while" loop, we could abbreviate this:

```
count = count + 3;
```

with this:

```
count += 3;
```

These shorter forms are quite handy in saving keystrokes when entering a long program, and they should be used when possible. Now let's take a look at another method of looping, the "for" statement.

The "for" Statement

The "for" statement, like the "while" statement, is a looping structure used to execute statements repeatedly within a program. The "for" statement specifies the number of times a statement or compound statement executes with a built-in counter. In addition, "for" allows you to initialize and perform regular updates on other variables. Its general syntax is:

```
for (init; test; change)
  statement1;
statement2;
```

where **init** is where your variable initializations are specified, **test** is where the determination to perform another iteration is located (similar to the test in a "while" loop), and **change** is where you describe your updates to variables that will be performed on each iteration. In its simplest form, a "for" loop might look like this:

```
for (num1 = 0; num1 < 5; num1++)
  printf("The current value of num1 is %d.\n", num1);
```

where **num1** is first set to zero, the check is made to determine if (num1 < 5) is true, and if so, the **printf** statement is executed. What do you think is displayed by this segment? If you guessed the numbers zero through four,

you are correct. What do you think would be displayed if we changed the third portion of the loop like this:

```
for (num1 = 0; num1 < 5; ++num1)
  printf("The current value of num1 is %d.\n", num1);
```

If you think the numbers one through five will be displayed, you are wrong! Although we know the pre-increment operator (as in ++num1) increments the variable before its value is used, in the "for" loop, the third (or update) portion of the statement does not get executed until after the first iteration. So, the results for this version of the "for" loop are identical to those of the previous one—zero through four.

What is the value of a loop? We have briefly discussed how a loop provides the programmer with additional control over the flow of a program. But a loop can be a real time-saver too. To illustrate, consider this program segment:

```
for (x = 1; x <= 100; x++)
  {
  scanf("%d",&num);
  printf("%d\n",num);
  }
```

When the program executes this loop, the values assigned to *x* are printed from the selected range of 1 to 100. In other words, the value of *x* is printed 100 times. An alternative method is simply to include 100 **scanf** and **printf** statements. Which method is more practical? Let's study a simple example:

/*This program performs simple accounting.*/

```
#include <stdio.h>
float sum, exp_sum, mth_net_inc, mth_exp;
float inc_ave, exp_ave;
int income, expenses;

main()
{
sum = 0;
exp_sum = 0;
printf("Enter your monthly income . . .\n");
for (income = 1; income <= 12; income++)
  {
  printf("Enter income for month %d: ", income);
```

```
  scanf("%f",&mth_net_inc);
  sum += mth_net_inc;
  }
printf("\n\nEnter your monthly expenses . . .\n");
for (expenses = 1; expenses <= 12; expenses++)
  {
  printf("Enter expenses for month %d: ", expenses);
  scanf("%f", &mth_exp);
  exp_sum += mth_exp;
  }
printf("\n\nYour total net income for the year is $%10.2f\n", sum);
printf("Your total expenses for the year is $%10.2f\n", exp_sum);
inc_ave = sum / 12;
exp_ave = exp_sum / 12;
printf("Your ave. monthly net income is $%10.2f\n", inc_ave);
printf("Your ave. monthly expenses is $%10.2f\n", exp_ave);
if (inc_ave < exp_ave)
  printf("Your accounts are in the red!\n");
else
  printf("Your accounts are in the black!\n");
getchar( );
}
```

When you execute this program, you are requested to enter your net income for each of the 12 months first and then each monthly expense for the same period. Examine the "for" statement itself. Both loops contain an ending value of 12, which specifies that each loop will execute exactly 12 times (1 to 12). The statement following the "for" statement is a compound set ({ and }). All statements contained within this block are executed 12 times. This is how we are able to request input via **scanf** 12 times. In addition, we included an accumulator within each loop to sum the amounts of net income and our monthly expenses. When each loop is finished, program flow exits the loop and executes the next available statement. Note that any range that counts 12 times can be used in this example. In other words, a range from 12 to 23 will loop 12 times as well. The beginning value and the ending value parameters of a loop that increments by one between these two values are what is important.

The inclusion of a compound statement following each "for" statement is important. Why? Because we want the I/O (**printf** and **scanf**) and the accumulator to execute 12 times. If the { and } pair is omitted, only the first statement following the "for" statement is executed as part of the loop. When

the loop is finished, program flow would pass to the **scanf**, which would only be executed once, and everything would be a mess!

You should be familiar with the components of the rest of this program. Our goal, however, was to simplify a programming task via the "for" statement, and although we have a fairly compact program (since we don't have 12 separate **printfs** and **scanfs** for the I/O), we could compact things even more like this:

```
/* This program performs simple accounting, but in less code */

    #include <stdio.h>

    float sum, exp_sum, mth_net_inc, mth_exp;
    int counter;

    main()
    {
    sum = 0;
    exp_sum = 0;
    printf("Enter your monthly income and expenses . . .\n");
    for (counter = 1; counter <= 12; counter++)
      {
      printf("Enter income for month %d: ", counter);
      scanf("%f",&mth_net_inc);
      sum += mth_net_inc;
      printf("Enter expenses for month %d: ", counter);
      scanf("%f",&mth_exp);
      exp_sum += mth_exp;}
    }
    printf("\n\nYour total net income for the year is $%10.2f\n", sum);
    printf("Your total expenses for the year is $%10.2f\n", exp_sum);
    printf("Your ave. monthly net income is $%10.2f\n", sum / 12);
    printf("Your ave. monthly expenses is $%10.2f\n", exp_sum / 12);
    if ((sum / 12) < (exp_sum / 12))
      printf("Your accounts are in the red!\n");
    else
      printf("Your accounts are in the black!\n");
    getchar();
    }
```

In this version, we have removed a couple of somewhat unnecessary variables: **inc_ave** and exp_ave. More importantly, however, is the fact that we were able to consolidate our income and expense loops into one. Here

we enter each month's income followed immediately by each month's expenses. Because of this, we were able to replace the loop counting variables **income** and **expenses** with the generic one, **counter**. In our haste to cut out variables and code, we actually added a bit of processing time to the program, since it now has to calculate both the average income (sum / 12) and the average expenses (exp_sum / 12) in both the **printf** statements and the "if" statement.

The "for" statement is a very flexible control structure. A program may contain several "for" statements within the same program block. In addition, we can write a program that specifies the number of iterations a "for" statement loops before it executes by predefining the ending value of the increment (e.g., 1 to 5). We can also write a program that lets the user interactively specify the number of iterations as the program executes (before the actual loop executes). The following example is another slightly modified version of our accountant program:

/*This program performs simple accounting, but in less code*/

```
#include <stdio.h>

float sum, exp_sum, mth_net_inc, mth_exp;
int num_mnths, counter;
main()
{
sum = 0;
exp_sum = 0;
printf("How many months do you wish to do: ");
scanf ("%d",&num_mnths);
printf("Enter your monthly income and expenses. . .\n");
for (counter = 1; counter <= num_mnths; counter++)
  {
  printf("Enter income for month %d: ", counter);
  scanf("%f",&mth_net_inc);
  sum += mth_net_inc;
  printf("Enter expenses for month %d: ", counter);
  scanf("%f",&mth_exp);
  exp_sum += mth_exp;}
}
printf("\n\nYour total net income for the year is $%10.2f\n", sum);
printf("Your total expenses for the year is $%10.2f\n", exp_sum);
printf("Your ave. monthly net income is $%10.2f\n", sum /
num_mnths);
```

```
printf("Your ave. monthly expenses is $%10.2f\n", exp_sum /
num_mnths);
if ((sum / num_mnths) < (exp_sum / num_mnths))
  printf("Your accounts are in the red!\n");
else
  printf("Your accounts are in the black!\n");
getchar( );
}
```

The primary difference between this program and the last version of the accountant program is that the user interactively selects the number of times the loop executes. This task is accomplished with the **scanf** statement, which assigns a value to the variable representing the ending value of the range in the "for" statement. Specifically, we used the variable **num_mnths** to represent the ending value. Therefore, from our previous program, the statement:

```
for (counter = 1; counter <= 12; counter++)
```

becomes:

```
for (counter = 1; counter <= num_mnths; counter++)
```

where the value assigned to the variable **num_mnths** is the ending value of the "for" loop that replaces the value 12. This value must be assigned before the "for" statement executes. In our example, the user is prompted to enter the desired value just before the "for" statement. Such a method provides a larger degree of flexibility. You can select 3 months, 6 months, 12 months, or any whole number that you desire.

Referring to the rest of the program, notice that the formats of the formulas for the **printf** statements that display the average value have been changed to incorporate the **num_mnths** variable. If we simply left these denominators at 12, the resulting values would only be correct when the user entered in 12 for the number of months to process.

Although we have not explicitly said so up to now, the "for" loop may be used to loop backwards like this:

```
for (i = 100; i > 0; i--)
  printf("We are looping backwards!!!\n");
```

This sort of "for" loop is very convenient to use in certain situations, but basically operates just like the upwards counting loops we have already seen.

That is, after initialization (i = 100), if the condition is true (i > 100), the associated block is executed, and each time the update portion (i--) is performed. In the next section, we discuss one more looping control structure, the "do-while" statement. This structure is the last of C's three looping options.

The "do-while" Statement

The "do-while" statement consists of two parts: the body and the termination condition. It is essentially an upside-down "while" statement in which the condition is not checked until the loop has performed at least one iteration. As with the "while" statement, the programmer should include a condition where the expression will be false and the flow exits the loop. Otherwise, the loop won't terminate, and we face the problem of an infinite loop. This simple program example satisifes this requirement:

/*This program calculates your family budget.*/

```
include (stdio.h)
float rent, utilities, groceries, car, misc;
float net_income, budget;

main()
{
printf("This program calculates your monthly family budget.\n");
printf("The program will quit when you are over budget.\n");
do
  {
  printf("Rent: ");
  scanf("%f",&rent);
  printf("Utilities: ");
  scanf("%f",&utilities);
  printf("Groceries: ");
  scanf("%f",&groceries);
  printf("Car payment: ");
  scanf("%f",&car);
  printf("Miscellaneous: ");
  scanf("%f",&misc);
  printf("Total net income: ");
  scanf("%f",&net_income);
```

```
  budget = net_income - (rent + utilities + groceries + car +
misc);
  printf("\n\nYou are left over with $%10.2f\n", budget);
  } while (budget > 0);
printf("\n\nYour expenses exceeded your net income!!!");
getchar( );
}
```

This program calculates a family's monthly budget. All the statements within the body of the loop will execute at least once. The program first instructs the user to enter a series of expenses (i.e., rent, utilities, groceries) followed by the user's net income. Expenses are totaled and then subtracted from net income to arrive at a budget for this particular month. If net income is greater than the expenses, the program repeats another calculation (calculate another month). The mechanism used to exit the loop is provided by the condition at the end of the loop (while (budget > 0)). Specifically, when the net income for a particular month is less than the total amount of expenses, program flow exits the loop and the program ends.

"while" vs. "do-while"

Because of the similarity in structure of the "while" loop and the "do-while" loop, these two loops are often compared. These two loops, however, have one significant difference regarding their execution. Both loops must test a condition in order to execute the loop itself. Performing the decision test differentiates between the two. Specifically, the "while" statement tests the loop condition before executing the body of the loop. On the other hand, the "do-while" statement tests the loop condition at the end of the loop. In other words, the "do-while" statement always executes its loop body at least once. The choice of which to use is entirely up to you, but for most instances, the "while" version will be the most appropriate.

Nested Loops

What is a nested loop? Nested loops are loops within a loop. We have illustrated nesting program structures before, such as the compound { and } pair and nested "if" statements. The logic for nesting a loop, such as the

"while," "do-while," or "for" statement is similar in execution. In simple terms, the way a nested loop works is that the outer loop executes its initial task, and then waits until the inner loop completes all of its loops or tasks. Here's a simple example that illustrates a nested "for" statement:

```
/*This program shows how nested loops work.*/

    #include <stdio.h>

    int loop1, loop2, cnt1, cnt2;

    main()
    {
    printf("How many times do you want the outer loop to execute: ");
    scanf("%d",&loop1);
    printf("How many times do you want the inner loop to execute: ");
    scanf("%d",&loop2);
    for (cnt1 = 1; cnt1 <= loop1; cnt1++)
      {
      printf("Outer loop iteration #%d\n", cnt1);
      for (cnt2 = 1; cnt2 <= loop2; cnt2++)
        printf("Inner loop iteration #%d\n", cnt2);
        }
    printf("\n\nAll loops are now complete.");
    getchar();
    }
```

This program requests users to select the number of times they want the nested loop to execute. First, you enter a value for the outer loop, and then you are requested to enter a value for the inner loop. In each case, the outer loop and inner loop will execute the number of times assigned to the ending value (loop1 and loop2) of the control variable (cnt1 and cnt2).

One More Loop: The goto Statement

As we discussed, the "while" and "do-while" statements are called conditional looping statements. In addition, we learned how to use the "if" and "switch" statements to perform a conditional branch. Conditional structures execute a selected number of statements depending on the value of an expression (the condition). C offers one additional looping structure that alters pro-

gram flow unconditionally. This type of unconditional branching is called the **goto statement**.

The goto statement is common among other programming languages, such as BASIC. However, it is not generally recommended because of its uncontrollable nature and its inconvenience when trying to trace program flow. Its purpose is to pass control of the program flow unconditionally from one point in a program to another, skipping the execution of any statement(s) between the two points. A common use of the goto statement is to exit from a loop.

The point where the flow of program control is transferred to is identified with a "label." The general format of the goto statement is:

```
goto label;
```

where "label" represents a valid name with naming rules identical to those of variables. For example:

```
here: printf("We used a goto statement\n");
```

"here" is a valid label and the statement:

```
goto here;
```

although it does not sound grammatically correct, causes program flow to jump to the **printf** statement denoted by the "here" label.

C offers several control structures designed for solid structured programming technique. The goto statement is not one of them. Programs can be written efficiently without the goto statement. We were tempted to omit this statement from our discussion; however, you do have the right to know it is available. Compare it to the control structures that have only one exit and one entry. Which structure would you like to read, follow, or debug? We think you will agree that you should avoid using the goto statement.

This chapter taught you several concepts of program loops. You learned about the three important looping structures—"while" loops, "for" statements, and "do-while" loops. You learned about counting loops and loops that accumulate. In addition, you learned about the differences between loops.

It's time to take another break. Review the summaries and exercises that follow. When you're ready, continue on to Chapter 5 to study a very important feature of structured programming in C—functions.

Glossary for Review

infinite loop—a loop that will never end because the condition to be tested never evaluates to a terminating value

pre-incrementing—adding one to the value of a variable before its value is used

post-incrementing—adding one to the value of a variable after its value is used

pre-decrementing—subtracting one from the value of a variable before its value is used

post-decrementing—subtracting one from the value of a variable after its value is used

Quiz

1. What is the primary difference between a "while" loop and a "do-while" loop?
2. What is the difference between num++ and ++num?
3. How many iterations will the following loop perform?

```
for (num = 5; num < 5; num++)
  printf("Hi there.\n");
```

Chapter 5

C Functions

In this chapter, you will learn:

- How to write a program that features structured design.
- What a function is, and how to define one.
- How to use a function correctly with variables and parameters.
- How to use the **return** statement.

Thus far, we have covered many important programming tools from which you can build useful programs. All of our examples have used a single program module or block to accomplish a single task. In this chapter, we are going to discuss ways of putting together a group of modules, each performing a single task. When the individual modules are taken collectively, a larger programming task is performed. This type of programming is called *modular programming*. One mechanism used to accomplish this goal is called a function.

Structured Design

Large programming problems are often broken down into smaller subprograms. Subprograms are further divided into smaller subprograms until each subprogram consists of a few manageable statements. These "miniprograms" are generally easier to solve than the original "macro" program. Each subprogram performs a particular task, such as processing data mathematically or printing results. The individual subprograms are treated as modules. The final program consists of a collection of these individual modules. This type of programming design is referred to as *structured* because of the hierarchical tree nature of modular programming.

In this chapter, we are going to discuss a particular type of structured design, called top-down design. *Top-down design* (sometimes called stepwise refinement) is a methodical approach where the original macro problem is represented by a main module. The main module could consist of the major programming steps required to solve a problem. The main module then calls upon the individual subprograms to solve specific tasks. When all the simpler tasks are solved, we have a solution to our original problem.

As stated, the main module is divided into smaller and smaller modules. Each module represents a specific level of programming that performs a task independently of the other modules. For example, the main module can be depicted as level 0, the next set of modules as level 1, and so on. Any module within this tree structure at a higher level can make demands on any module at a lower level.

Writing programs as a collection of modules is a very important feature of the C language. Let's now turn our attention to the logical method used to identify a subprogram or module—the function.

What Is a Function?

A *function* is used to identify a collection of C statements that perform a specific task. The function represents a subprogram and is executed each time the function is invoked. To access a function, you assign it a name and then "call" it by name from some other point in the program. Once a function has executed, the flow of program control returns to the calling routine. For example, we have already worked with the functions **printf** and **scanf** (to name but two). Although they may appear to you to be some magical black box in which you send some information and get other information back (in the form of a display, assignment, etc.), they are simply C functions that, when invoked, cause program flow to start executing their statements. Program flow is then resumed with the statement that immediately follows the one that called the function.

What Does a Function Consist Of?

The structure of the function is just like the program examples that we have been discussing. A function consists of two parts: the heading and the body. The heading, or name assigned to a function, is simply the label you use to refer to the function. The body is the sequence of program statements (including declarations) specific to a function. Generally, there are no restrictions placed on the number of statements that a function may contain. However, one of the primary purposes of using functions is to break down the programming problem into more meaningful tasks for both readability and to reduce the number of identical code blocks. For these reasons, you really don't want to have a function that is ridiculously long and could be broken down further. The function may look just like a miniprogram, complete with a section for declarations and a { and } pair to show the beginning and ending points of the function.

Where Is a Function Placed?

In general, a function may be placed anywhere in your source code in relationship to the main routine. Unlike languages such as PASCAL, the func-

tions do not have to be declared before the main routine, so it is up to you to decide where you would like to place them. We have seen code that places main at the beginning of the source and at the end. While neither may be more correct, we prefer to place main at the beginning.

How Are Functions Used?

Suppose you have a program that you wish to perform a number of statements over and over again. We can simply rewrite the same sequential set of statements each time they are required; or, we can "call" a miniprogram or function to accomplish the same task. When the task is completed, we then continue on about our business. For example, let's examine the following program. It contains two mathematical functions, one for addition and one for multiplication:

```
/*This program shows how functions may be used.*/

    #include <stdio.h>

    float num1, num2, sum, product;

    main()
    {
    printf("Enter a number: ");
    scanf("%f",&num1);
    printf("Enter another number: ");
    scanf("%f",&num2);
    printf("\n\nWe are now computing the addition . . . \n");
    add();
    printf("\n\nWe are now computing the multiplication . . . \n");
    mult();
    printf("\n\nThe program is finished!");
    getchar();
    }

    add()
    {
    sum = num1 + num2;
    printf("The sum of your numbers is %f.\n", sum);
    }
```

```
mult()
{
product = num1 * num2;
printf("The product of your numbers is %f.\n", product);
}
```

We think it's important that you understand the flow of program control in our simple example above. Let's take a look step by step.

When you execute the program above, you are asked to input two numbers. This is accomplished by the two **printf** and **scanf** combinations in the main routine. Next, a message is displayed that informs you that the addition is about to be performed. At this point, the routine **add()** is invoked, and control jumps to this function. The **add()** routine has all of its statements enclosed with a { and } pair. The first statement executed is the calculation of the sum. Next, the sum is printed out, and the function ends. Then, control returns to the main routine, and a message is displayed indicating that the multiplication is about to be performed. As you can see, when we left the main routine, we were about to perform this **printf** statement, and as soon as we return, it is executed. At this point, the **mult()** function is invoked, and control is passed to it, just as it was to **add()**. First the product is calculated, and then it is displayed. This function then ends, and control is transferred back to the main routine, where the message "The program is finished!" is displayed.

Two important points to note are that the names of the functions **add()** and **mult()** appear at the beginning of those functions just as **main()** appears at the beginning of the main routine. In addition, note again that the function's statements are enclosed with a { and } end pair, just like those of the main routine, and that compound statements would appear within braces just like they have in **main()**.

Writing programs, such as the one just discussed, is easily accomplished using top-down design. The first step in writing the actual program code involves breaking down the main program into smaller manageable miniprograms. The logic of the main program is first developed. In our previous example, for instance, we put all the little initializing input/output statements in the body of the main program. Next, we break the building blocks of the program into separate smaller tasks, such as a function to do addition and its output, another to do multiplication and its output, etc. Whenever such a task is required, we simply call on that particular procedure again. Remember, we can call a function from any point in the main program as many times as we want. We can even have one function call another function. This should

be easy to visualize, since **main( )** is nothing more than a special function itself.

Obviously, a function may contain a miniprogram that completes a more complex task than adding or multiplying two numbers. However, the concept is the same. It is important that you understand the logic behind structured programming with a top-down design where large programs are really a collection of smaller ones. Later in this book, we will introduce applications that utilize this very important feature.

Using Variables with Functions

Understanding and defining how a procedure works is a simple concept. However, there are a few additional rules that we need to discuss. In our previous example, we created a program that transmitted information from a procedure back to the main program. All of our variables were global to the program, and so any function was permitted to modify them at any time (recall the global variable description in Chapter 2). The way we modified these variables via the functions **add( )** and **mult( )** was via their global nature. *However, variables can be passed from one function to another, and* therefore they need not be global to the program in order to be modified.There are actually two different methods of passing values to a function—by value and by address. These two methods differ greatly and will be discussed later. First, let's review a bit and further discuss the concept of global versus local variables.

Global vs. Local Variables

Before you write a function that uses variables, you must first determine how it's going to be used and what effect the variable will have on the rest of the program. In C, there are rules for governing the variables in a function, which are called *scope* rules. The scope of a variable is defining its accessibility to the rest of the program. For example, if a variable named **a** is used in a function, do you want its value accessible by other functions? Our mathematics program used this type of variable, a *global* variable. A global variable has meaning throughout the program.

On the other hand, suppose you want a function that uses a variable that has no effect outside the function itself. This type of variable is called a *local*

variable. A local variable is one that only has meaning while the body of the function where it resides is executing. When execution of this function is complete, a local variable no longer exists (unless it is declared to be a static variable).

How do we identify or define the scope of a variable? As you recall, all variables must first be declared before they are used in a program. This rule applies whether the variable is used in the body of the main function or within another function. It's where you place the variable declaration that defines its scope. If the variable is declared within a function's { and } block, it may be accessed only by that function. If however, the variable is declared outside the main's { and } block (as our variables were), then it is accessible by the entire program.

Beware if you declare a variable with the same name both globally and locally to a function. Why? Because once the function transfers control back to the calling routine, the local variable will no longer exist as far as the calling routine is concerned. The value assigned to the variable will be the last value assigned from within the rest of the program that addressed the main declaration. The result may not be what you expected.

When do we use a local variable or a global one? Good programming practice dictates that, whenever possible, you use a local variable instead of a global one. Why? This is a practical question, since our first example used global variables. However, a global variable can sometimes wreck a good program. The reason is simple. It's wise to use a function to complete its task, and then go about your business with the rest of the calling routine. In a large program, you may wish to use the same variable used within a function again. A global variable retains its value, and can produce undesirable side effects later in your program. A local variable, however, has no effect on main program and can be used over and over again without fear of unexpected results.

Perhaps the best reason to use a local variable in a function is the flexibility when making multiple calls to the same routine. Each time a call to a function is made in such a case (again, assuming no static variables), the variables will be starting from scratch. If global variables were used, the values left over from the previous execution of the function would still be in effect.

Regardless of the reason, most experienced C programmers agree that unnecessary global variables often lead to programming bugs. Such bugs are difficult to find even for the most seasoned programmer. As a rule of thumb, you should try to use local variables whenever possible to avoid the problems outlined above.

We just made a case for using local variables, but we haven't defined the method for exchanging information to and from a function when local variables are used. To do this, we use a mechanism called a parameter.

Value Parameters

Calling a function using variable parameters is slightly different than calling a function without them. Suppose we want to pass the values of the variables **num1** and **num2** from the main routine to the routines **add()** and **mult()**. Our program would look like this:

```
/*Now we are passing parameters by value*/

#include <stdio.h>

main()
{
float num1, num2;

printf("Enter a number: ");
scanf("%f",&num1);
printf("Enter another number: ");
scanf("%f",&num2);
printf("\n\nWe are now computing the addition . . . \n");
add(num1, num2);
printf("\n\nWe are now computing the multiplication . . . \n");
mult(num1, num2);
printf("\n\nThe program is finished!");
getchar();
}

add(x, y)

float x, y;

{
float sum;

sum = x + y;
printf("The sum of your numbers is %f.\n", sum);
}

mult(x, y)
```

```
float x, y;

{
float product;
product = x * y;
printf("The product of your numbers is %f\n", product);
}
```

In our new program, you should notice that the variables **num1** and **num2** are now local to and accessible only by **main( )**. Also, the functions **add( )** and **mult( )** now have parameters associated with them in both their calls and definitions. For instance,

```
add(x, y);
```

declares **add( )** as a routine with two parameters, **x** and **y**, whose type are float as defined by the next statement:

```
float x, y;
```

Whenever you declare a function to have parameters, you must define them after this declaration.

Now when we invoke **add( )**, we place the values of **num1** and **num2** within the parentheses in the call. This is identical to the way we have been calling **printf** and **scanf** all along; we simply place the parameters in the order in which the function expects them. The same rules outlined for **add( )** apply to the function **mult( )**. You should also notice that the variables **sum** and **product** are no longer declared globally either. They are now local variables to the routines in which they are used. That is, now only **mult( )** can access the variable **product**, and **add( )** is the only function that can access the variable **sum**.

We refer to this method of parameter passing as passing by "value," primarily because the parameters that are passed may be changed in the called function, but the calling function will not see any change. For instance, if we were to change **x** or **y** in **add( )** or **mult( )**, only these local versions of the parameters **num1** and **num2** would be affected. The actual parameters **num1** and **num2** would remain unchanged. Later we will see a method of changing parameter values. But first let's look at how a function can return a single value.

The Return Statement

A function may be set up to return one value to its calling function via the **return** statement. The **return** statement's syntax is:

```
return(value);
```

where the value in parentheses is returned to the calling routine. For example, try this new version of our mathematics program:

```
/*Now we are using the return statement.*/

#include <stdio.h>

float num1, num2, sum, product;
float mult(), add();

main()
{
printf("Enter a number: ");
scanf("%f",&num1);
printf("Enter another number: ");
scanf("%f",&num2);
printf("\n\nWe are now computing the addition . . . \n");
sum = add();
printf("The sum of your numbers is: %f.\n", sum);
printf("\n\nWe are now computing the multiplication . . . \n");
product = mult();
printf("The product of your numbers is: %f.\n", product);
printf("\n\nThe program is finished!");
getchar();
}

float add()
{
float x;

x = num1 + num2;
return(x);
}

float mult()
```

```
{
float x;

x = num1 * num2;
return(x);
}
```

You should first of all notice that we made **num1**, **num2**, **sum**, and **product** global again, and our next declaration is something new. We have declared our function **add( )** and **mult( )** to be of type float. This simply means that they will be returning a value (via the **return** statement) of type float. The return value of a function defaults to int, and we would not have had to make these declarations if **add( )** and **mult( )** were returning int values. Nevertheless, it is good practice always to declare your functions like this even if they are returning an int value.

The first several statements of **main( )** are unchanged until the statement:

```
sum = add( );
```

which means that the function **add( )** is invoked, and the value it returns is to be placed in the variable sum. Looking at **add( )**, the local variable **x** is declared, and the sum of **num1** and **num2** is placed in this local variable and is returned to **main( )** by the statement:

```
return(x);
```

It's as simple as that! The routine **mult( )** is invoked and a value returned in the same manner as **add( )**. The variables **sum** and **product** are then used in **printf** statements after the invocations of **add( )** and **mult( )**, respectively.

The **return** statement is not only responsible for sending a value back to the calling routine, its execution also terminates its routine's execution. For example, if we were to change **add( )** to look like this:

```
float add( );
{
float x;

x = num1 + num2;
return(x);
printf("Hi there.\n");
}
```

the **printf** statement would never be executed. As soon as the **return** statement is reached, no other statements within that function will execute. This can be advantageous to the programmer who wants to exit routines from several different points, but in general, it is analogous to the **goto** statement in terms of unconditionally jumping out of a piece of code. And you know how we feel about **gotos**! As a final note on return, the value portion of the general syntax is optional. So, you could use a **return** statement that looks like this:

```
return;
```

and returns no value but does cause control to be passed back to the calling function.

Now that we have seen how to pass information by value and with the **return** statement, let's look at how to pass parameters by address.

Variable Parameters

When we pass a variable parameter in a function, it is usually just a copy of that variable, and the calling function may not use this to change the value of the actual variable. We say "usually" because all variables except arrays (i.e., strings) are passed by value unless they are preceded by the address operator **&**. We have already seen this operator used in **scanf** statements when we are inputting certain values. These certain values that require the **&** in **scanf** are, not coincidentally, any variable except arrays. When a string variable (which is the only array type we know so far) is passed to a function (such as **scanf**), a pointer to the beginning of it is passed, not the actual string. This is done automatically by C with no intervention on your part. Because of this, the called function is able to modify the original variable, since it knows exactly where it resides in memory. All other parameters are not passed as addresses by default. However, their addresses may be passed by using the **&** operator if it is desired. Consider our final version of the mathematics program:

```
/*Now we are using pointers to variables.*/

#include <stdio.h>

main()
```

```
{
float num1, num2, sum, product;
printf("Enter a number: ");
scanf("%f",&num1);
printf("Enter another number: ");
scanf("%f",&num2);
printf("\n\nWe are now computing the addition . . . \n");
add(num1, num2, &sum);
printf("The sum of the numbers is: %f.\n", sum);
printf("\n\nWe are now computing the multiplication . . . \n");
mult(num1, num2, &product);
printf("The product of the numbers is: %f.\n", product);
printf("\n\nThe program is finished!");
getchar( );
}

add(x, y, z)
float x, y, *z;

{
*z = x + y;
}

mult(x, y, z)
float x, y, *z;

{
*z = x * y;
}
```

Again, there are no new items in the program until we reach the invocation for the **add** function:

```
add(num1, num2, &sum);
```

This is read to say that "**add** is invoked and the values of **num1** and **num2** are passed along with the address of the variable **sum**." The declaration of **add**( ) shows that it has three parameters: **x**, **y**, and **z**. These parameters are then defined (just as they were in our value parameter example) by specifying their type and names. However, the last variable has an asterisk in front of it. Why? Because this parameter is a pointer and * is the pointer operator. You would read this declaration as "**add** is a function with three parameters: **x** and **y** are float values, and **z** is a pointer to a float value." Remember that this mysterious third parameter has an address in it and is therefore called a

pointer to a value. With this address, we are able to take the sum of **x** and **y** and put the result where **z** points. That is the meaning of:

```
*z = x + y;
```

In short, **z** points to the variable **sum**, so all reference to ***z** are actually references to **sum** itself. The concepts behind the **mult()** function are identical to those of **add()**, except that **z** now points to the variable **product**.

In summary, you just completed a very important chapter on C programming. You learned how to define and to write a function properly. You now know the difference between a global variable and a local variable. You learned how to pass information using value and address parameters using pointers. Lastly, you were also introduced to the **return** statement. Take a few minutes, give yourself a pat on the back, and review what you have learned. You'll discover you can write some interesting programs with what you have learned thus far.

Glossary for Review

modular programming—the breaking down of various tasks within a program into individual functions as opposed to one large routine

top-down design—a design where the resulting program has one main driving routine that calls other routines to perform specific tasks

scope rules—the rules governing which functions can access which variables (i.e., global, local)

Quiz

1. What is the difference between global and local variables?
2. What does this declare:

   ```
   int *number;     ?
   ```

3. What is the difference between passing values by value and passing addresses of values?

Chapter 6

Library Features

In this chapter, you will learn:

- What the library concept is.
- How to use the available library routines for mathematics, strings, etc.
- How to use the numerous GEM library routines.
- How to create a window in C.

How to Avoid Reinventing the Wheel

When you begin writing your own programs, you soon realize that it would be nice if the computer could magically perform some common tasks that you use in different programs. For example, you might want to write a routine that would allow you to read in a string containing multiple words. The standard **scanf** function will simply truncate all the data entered after a "white space" (e.g., tab, space, etc.), so you must have some method of not terminating the input when a white space is encountered. One way would be to use the **getch( )**function repeatedly until a new line is input. Each successive character could be placed into a character string, and the new line would signal us to place a **'\0'** terminator at the end of the string.

It may have only taken you a few minutes to develop a function like this, but if someone had already written one for you, this would be a waste of time. In fact, there is a function available to you in the Standard I/O C library that is very similar to our description. But before we discuss it and several other library routines, let's first take a closer look at what is meant by a library.

The Central Library

If you're an avid book reader, you have come to learn that your local library has a wealth of books on topics from A to Z. And you've learned that by taking advantage of this library's offerings you may have saved yourself hundreds of dollars by not having to purchase these books yourself. The C libraries are very similar to your local library in that they hold a wealth of functions enabling you to perform various tasks quickly, from complex mathematics to high-resolution graphics. Because of this, the C libraries can save you time (just as your library has saved you money), since you don't have to write these commonly used routines yourself. All you need to do is properly invoke them and follow a couple of simple rules: 1) if the library routine returns a value, make sure that the variable you are assigning to its value is compatible with what the function returns; and 2) if the routine passes parameters, make certain that the order of your parameters is identical to that of the routine and that their types and sizes (e.g., long, short) match.

Once you have an understanding of the routines, all you need to do is

use the **#include** directive to let the compiler know you will be using that particular library and reference them as if they were your own. The remainder of this chapter is dedicated to presenting some of the more popular routines and macros available, with a brief discussion of each.

Standard C Functions

Mathematical

The Cosine Function. This is a trigonometric function available to you that returns the cosine of the angle expressed in radians with this format:

```
result = cos(angle);
```

where both **result** and **angle** are declared as double.

The Sine Function. This is another of the trigonometric functions available to you that returns the sine of an angle expressed in radians with the following format:

```
result = sin(angle);
```

again, where both **result** and **angle** are double.

The Tangent Function. Again, this is a trigonometric function in which the result and the angle are double with the following format:

```
result = tan(angle);
```

The Square Root Function. Use this function to determine the square root of a numeric value (i.e., the square root of 4 is 2) with this format:

```
result = sqrt(number);
```

where both **result** and **number** are double.

The ASCII-to-Integer Function. If for some reason you would desire to read

in a number as a string and then convert it to an integer, this function will do the job:

```
number = atoi(string);
```

where **number** is an integer, and **string** may either be the name of a character array or a pointer to a character string. The reason for this leniency for **string**'s type is that a character array's name is viewed internally by the compiler as a pointer to the first element of the array.

String

The gets Function. Up to now the only way we know of to read in a character string is with the **scanf** function like this:

```
scanf("%s",&a_string);
```

A serious shortcoming of this method is the fact that any white space in the input string is viewed by **scanf** as the end of the string. Therefore it and all the characters following it are ignored. In short, **scanf** keeps reading in characters until it encounters either a white space or a new line and then replaces that character with the null terminator (**'\0'**). The easiest way of avoiding this shortcoming of **scanf** is to use the **gets** function with the following format:

```
gets(a_string);
```

The **gets** function will allow you to read in, say, a last and first name and place them both in the same string variable without any further manipulation. [NOTE: For Lattice C users, a problem with **gets** was discovered by the author in an early version (3.03.03). If you used a **gets** after a **scanf** in a program, the **gets** would usually be ignored as if the input buffer was not completely cleared out after the **scanf**. A null value would then be placed in the variable passed to **gets**, resulting in a variable that never held much of anything! This problem has been reported to Metacomco, who said they would look into it.]

The strcat Function. The ability to concatenate strings, or place one string at the end of another, is handled by the **strcat** function:

```
strcat(a_string,b_string);
```

where **a_string** and **b_string** are both pointers to strings. The **b_string** is concatenated to the **a_string** so that, if **a_string** contained "not" and **b_string** contained "hing" before the call to **strcat**, the value of **a_string** after **strcat** would be "nothing." Notice that there are no spaces implicit in the call to **strcat**, so if you concatenate a first and last name, be sure to have a space either at the end of the first name or at the beginning of the last name before calling **strcat**.

The strcmp Function. In order to compare one string to another, you cannot simply compare the string names like this:

```
if (a_string == b_string)
```

because this compares two unique addresses and will never be true (remember again that the name of a string simply represents the address of the first character of the string!). The **strcmp** function may be used instead:

```
strcmp(a_string,b_string);
```

This function will return a value of zero if the strings are equal, a negative number if **a_string** is less than **b_string**, or a positive number if **b_string** is less than **a_string**. Be sure to use this function always when comparing strings: If you see an "if" statement like the one above, don't be surprised if it is always evaluated as being false.

The strlen Function. In other languages, the length of a string is specified in the first one or two bytes of the string so that the end point is known. C does not need this format, since it uses the null terminator to signify the end of a string. Sometime, however, you might need to know the length of a particular string, and rather than write a routine that will count characters until a null byte is found, you could use the **strlen** function:

```
strlen(a_string);
```

This function returns an integer value that represents the length of the string up to but not including the null terminator.

Macros

Also available to the C programmer are a number of very useful macros. The following is a list of several of these, along with a description of each:

isalpha(i)—this macro will return a nonzero value if **i** is an alphabetic character (e.g., *A* through *Z* or *a* through *z*); a zero will be returned otherwise.

isupper(i)—a nonzero value is returned if **i** is an uppercase character (e.g., *A* through *Z*); a zero is returned otherwise.

islower(i)—a nonzero value is returned if **i** is a lowercase character (e.g., *a* through *z*); a zero is returned otherwise.

isdigit(i)—a nonzero value is returned if **i** is a numeric character (e.g., 0 through 9); a zero is returned otherwise.

isspace(i)—a nonzero value is returned if **i** is a white space character (e.g., space, tab, etc.); a zero is returned otherwise.

isalnum(i)—a nonzero value is returned if **i** is either an alphabetic or numeric character (i.e., fitting the criteria of either **isalpha(i)** or **isdigit(i)**, above); a zero is returned otherwise.

isprint(i)—a nonzero value is returned if **i** is a printable character; a zero is returned otherwise.

iscntrl(i)—a nonzero value is returned if **i** is a control character; a zero is returned otherwise.

isascii(i)—a nonzero value is returned if **i** is an ASCII character (decimal values 0 through 127); a zero is returned otherwise.

toupper(i)—if **i** is a lowercase character, it is converted to uppercase.

tolower(i)—if **i** is an uppercase character, it is converted to lowercase.

abs(i)—the absolute value or magnitude of **i** is returned (e.g., abs(9) is 9, abs(−9) is 9, etc.).

GEM Functions

Included with your C compiler is a fairly extensive interface to the GEM, or Graphics Environment Manager, routines, which enable you to make your applications easily run like other programmers' in both look and feel. The Megamax C package has descriptions of each of the GEM and AES routines that they support. These descriptions include explanations of the routines as well as fairly thorough explanations of the parameters that are passed to them.

As a C programmer on the ST, you will probably never use most of the routines contained in this GEM library, but for when you do, if you are using Megamax C you need not worry about having to pull in special libraries; the

shell takes care of this for you by checking all the libraries on your System disk. If you are using another compiler, however, say, Lattice C, all you need to do is use the **#include** directive for the **gemlib.h file** (if you intend to use any of the constants within it), change your linkage control file to **LIBRARY the gemlib.bin file**, and call the routine just as you would any other function (see the description of the linkage control file in your Lattice C manual). Be extra careful in the sizes of the parameters that you pass any of these routines! This is a rule that should always be followed no matter which library you are working with. But remember, if the routine is looking for a short or a word parameter, don't pass it an int or a long.

The remainder of this section is a description of a few GEM routines that are either necessary or very useful for your applications. Take note that if you are wishing to learn GEM graphics routines, they will be covered in Chapter 10; these routines are more general purpose ones that everyone should be interested in.

appl_init( ) Before you can call any other GEM routines in your program, you must call the **appl_init** function. The basic premise behind this is initially to identify yourself. **Appl_init** is also used to allow multitasking, or the running of more than one application at once. For this reason, **appl_init** will return a long value that represents your application identification number as follows:

```
my_id = appl_init();
```

If **my_id** is not greater than zero, the function did not work. For our purposes, it is not necessary to look at the value returned from **appl_init**.

graf_handle( ) The next step in initializing your program is to get a chunk of memory that will contain all the attributes of the application. The **graf_handle** routine will return a pointer to this area that later may be used as a parameter to other routines specifying our application. We have always used **graf_handle** like this:

```
handle = graf_handle(&dummy,&dummy,&dummy,&dummy);
```

The four parameters represent character attributes that we do not wish to change from the default values. Now that we have a handle to call our own, we can use it to open our own workstation and set more attributes.

v_opnvwk( ) The **v_opnvwk** function is invoked like this:

```
v_opnvwk(work_in,&work_stn_hndle,work_out);
```

where **work_in** is an 11-element array containing information such as line type, text color, fill type, etc.; **&work_stn_hndle** is the address of our handle from **graf_handle**; and **work_out** is an array that is filled in by **v_opnvwk** and contains advanced information that we don't use in this book, such as raster information. We have always initialized the **work_in** array to 10 ones and a 2 as follows:

```
for (loop = 0; loop < 10; loop++) work_in[loop] = 1;
work_in[10] = 2;
```

This is a fairly standard way of setting up these attributes and should suffice for your programs.

As was mentioned, we don't ever look at the **work_out** array, but, of course, it must be present in your call to **v_opnvwk**. Finally, we can see the first use of our **work_stn_hndle** from **graf_handle** here as it specifies our application to GEM.

v_clsvwk( ) Once we are finished with our application, we must call **v_clsvwk** in order to close our workstation. The function has one parameter, the handle to our workstation, and is called as follows:

```
v_clsvwk(work_stn_hndle);
```

appl_exit( ) The last task to be completed in a simple program is to tell GEM that we are finished by calling **appl_exit**:

```
appl_exit( );
```

This lets GEM know that the application will now be terminated and any overhead variable space can be released.

graf_mouse( ) When you want to change the characteristics of the mouse, **graf_mouse** may be called:

```
graf_mouse(shape,&nothing);
```

where **shape** is a long value that defines the mouse form, and **nothing** is an address of a custom mouse's shape's description in memory. We have used **graf_mouse** to turn the mouse's display on and off using two of the constants defined in the **gemlib.h file**: **M_OFF** and **M_ON**. There are several other constants that you can use to change the mouse, such as **ARROW** (the familiar arrow pointer), **TEXT_CRSR** (the I-Bar form used in certain word processors and editors), etc. If the **USER_DEF** constant is used, **graf_mouse** will go to the address of **nothing** and use the values there to build a user-defined mouse form.

v_clrwk( ) If you want to clear the screen in your C program, call **v_clrwk** as follows:

```
v_clrwk(work_stn_hndle);
```

Again, we need to use the **work_stn_hndle** in order to specify which workstation we are referring to. However, a call to **v_clrwk** is not enough to clear the screen in the traditional sense; the cursor is still located in its last position, so we also need to move the cursor to the home position with **v_curhome**.

v_curhome( ) After calling **v_clrwk**, **v_curhome** should be invoked to move the cursor to the home position:

```
v_curhome(work_stn_hndle);
```

The combination of these two routines will clear the screen and place the cursor in the home position much like a clearing routine in other languages (e.g., CLS in many forms of BASIC). [NOTE: As of release 1.0 of Megamax C, their **v_curhome** is not implemented; perhaps Megamax will be including this function in release 2.0.]

Windows

Probably the most noticeable feature of the GEM interface for the Atari 520 ST is windowing. The ability to manipulate multiple windows simultaneously on the screen is certainly an attribute that adds to the user-friendliness of the system. However, if you're used to working with functions to control windows on, say, the Apple Macintosh, you'll be greatly disappointed by the amount of code necessary simply to create a window, let alone direct I/O to

it; while it requires only a handful of instructions to display and write to a window on the Macintosh, it takes several dozen instructions to do the same on the ST. As you have seen in the program examples up to this point, it's not necessary to work with windows at all on the ST. For this reason, and the fact that an entire book could be written on window handling, we will simply show you how to create and display a window in C.

The following is a C listing that, when compiled, linked, and executed with a PRG extension, will display the outline of a window with corner coordinates (50,50), (550,50), (50,300), and (550,300). The window will be displayed until the RETURN key is pressed on the keyboard.

/*Window Display Program*/

```
#include <stdio.h>
#include <gembind.h>
int wind_handle;        /*pointer to our window*/
int work_stn_hndle,handle;
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

main()
{
int nothing;

window_set_up();
graf_mouse(M_OFF,&nothing);
getchar();        /*wait for user to press RETURN to quit*/
graf_mouse(M_ON,&nothing);
clean_up();
}

window_set_up()
{
short loop;
int work_in[11];
int work_out[57];
intt dummy;
int x_loc = 50;
int y_loc = 50;
int height = 250;
int width = 500;

appl_init();
handle = graf_handle(&dummy,&dummy,&dummy,&dummy);
work_stn_hndle = handle;
```

```
for (loop = 0; loop < 10; loop++) work_in[loop] = 1;
work_in[10] = 2;
v_opnvwk(work_in,&work_stn_hndle,work_out);
wind_handle = wind_create(0x02,x_loc,y_loc,width,height); /*get
handle*/
wind_open(wind_handle,x_loc,y_loc,width,height); /* open the
window.*/
}

clean_up()
{
wind_close(wind_handle); /*close the window*/
wind_delete(wind_handle); /*get rid of the window*/
v_clsvwk(work_stn_hndle);
appl_exit();
}
```

You should pay close attention to this program's format because it is the way the remaining chapters in this book are laid out. In order to access many of GEM's abilities, we use a setup routine that invokes **appl_init()** and **v_opnvwk()**, which were described above. In addition, when we are finished with our program, we call a **clean_up()** routine that closes our workstation via **v_clsvwk()** and ends the application via **appl_exit()**. After the program initialization, a call to the **gemlib** routine **wind_create** is made as follows:

```
wind_handle = wind_create(0x02,x_loc,y_loc,width,height);
```

where **0x02** is the value passed for the window flags, **x_loc** and **y_loc** are the coordinates of the upper left-hand corner of the window, and **width** and **height** specify how wide and high the window extends. These points are based on the screen's monochrome coordinate system, which has 640 points horizontally and 400 points vertically. Each point is also commonly referred to as a picture element, or pixel. So there are 256,000 pixels in monochrome mode.

The possible flags that may be set include the following bit positions:

0	title bar and name
1	symbol to close
2	symbol to make the window full-size
3	symbol to move the window

4	information line
5	symbol to resize the window
6	up arrow symbol
7	down arrow symbol
8	vertical slider
9	left arrow symbol
10	right arrow symbol
11	horizontal slider

In our example, only bit number 1 is set, so that the only extra feature of the window is a close symbol. The previously initialized variables **x_loc**, **y_loc**, **height**, and **width** are also passed to **wind_create** to specify the maximum size of the window. A handle for the window is returned by **wind_create** and placed in the variable **wind_handle**. This handle, similar to our familiar **work_stn_hndle**, is simply a pointer to the information that describes our window; it must be used in any further calls to window routines to specify this window.

Next, the **wind_open** routine is invoked to display the window. The parameters passed to it are the **wind_handle** (which was set by **wind_create**, above), the coordinates for the top left-hand corner of the window, as well as its width and height. For simplicity, we used the same values for the last four parameters that we used in **wind_create**; but you could have used different values, which would have resulted in a smaller window size.

The **getchar()** routine is then called to await the depression of RETURN, and finally the **clean_up()** routine is invoked first to close the window via the **wind_close** function and then to delete the window with **wind_delete**. Each of these routines takes the window handle as an argument in order to know which window you want to close and delete. Once you close a window you may open it again later, but once you have deleted the window, its attributes no longer exist in memory.

You might have noticed that we declared five int variables that we never used in the line:

```
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];
```

These variables are used in some of the VDI routines on the ST. If you are using any VDI routines, these variables must be declared in your program. You don't need to know what they are responsible for, but you will run into linkage errors on Megamax C if you do not include them.

As was mentioned above, this is a fairly simple look at windowing on the ST, and you could easily develop very sophisticated applications without ever even having to use windows at all. However, if you do need more information on ST window routines, we suggest that you take a look at the Atari ST GEM Programmer's Reference Guide from Abacus Software; this book goes into painful detail on advanced window concepts as well as event handling, which is a must for any serious window application.

Glossary for Review

library—a central gathering of commonly used routines; the various C libraries (e.g., **gemlib.bin**) have routines for mathematics, string manipulation, graphics, etc.

pixels—the tiny dots that make up the screen of a computer display; the ST has about 256,000 of them!

Quiz

1. What are the results of the following functions/macros?

A. **strcat("Ata","ri");**
B. **sqrt(25);**
C. **abs(12);**
D. **islower('a');**
E. **strcmp("Joe","Joe");**
F. **isspace('3');**

2. What happens if **v_clrwk** is not followed by a call to **v_curhome**?

3. What is the difference between the **x_loc**, **y_loc**, **width**, and **height** parameters in **wind_create** and **wind_open**?

Chapter 7

Programmer's Corner

In this chapter, you will learn:

- How complete C programs operate on the Atari ST.
- How to modify existing programs to suit your particular needs better.
- Why comments are such a critical part of programming.
- The importance of proper error handling.
- Four programs dealing with metric conversions, guessing numbers, hexadecimal notation, and album-to-cassette length calculation, which are all documented with line-by-line and variable explanations.

The Entire Picture: Complete C Programs

Now that you have learned most of the individual elements of C programming, it is time to put everything together and discuss a few application programs. Each of these programs has been selected and designed for the purpose of tying together all the topics that have already been discussed, as well as preparing you for further study in the areas of file handling, graphics, recursion, and arrays. Let's first take a look at a program that will perform various conversions to the metric system.

Metric Conversions Program

Enter the following program into your ST while paying close attention to spacing within quotes and the use of special characters (e.g., &, #, etc.):

/*Metric Conversion Program*/

```
#include <stdio.h>
#include <gembind.h>

#define METER_CONV 39.37 /*conversion constant of inch to
meter*/
#define LITER_CONV 61.02 /*conversion constant of cub. inch to
meter*/
#define GRAM_CONV 0.035 /*conversion constant of ounces to
grams*/

short work_stn_hndle,handle;
int nothing;
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

main()
{
int done;       /*are we finished yet?*/
int valid;      /*is this valid input?*/
char choice;    /*user's choice from menu*/
```

```
window_set_up( );
graf_mouse(M_OFF,&nothing);
done = FALSE;
while (!done)
  {
  valid = FALSE;
  while (!valid)
    {
    printf("Which do you wish to do:\n\n");
    printf("  1. Convert from inches to meters\n");
    printf("  2. Convert from cubic inches to liters\n");
    printf("  3. Convert from ounces to grams\n");
      printf("  4. QUIT\n\n");
      printf("PLEASE ENTER 1, 2, 3 OR 4");
      switch (chice = getchar( ))
        {
        case '1': inch_to_meter( );
                  valid = TRUE;
                  break;
        case '2': cub_inch_to_liter( );
                  valid = TRUE;
                  break;
        case '3': ounce_to_gram( );
                  valid = TRUE;
                  break;
        case '4': valid = TRUE;
                  done = TRUE;
                  break;
        default:  printf("\nINVALID RESPONSE . . . PLEASE TRY
AGAIN!\n\n");
        }
      }
      }
graf_mouse(M_ON,&nothing);
clean_up( );
}

window_set_up( )
{
short loop;
int work_in[11];
int work_out[57];
int dummy;
```

```
appl_init( );
handle = graf_handle(&dummy,&dummy,&dummy,&dummy);
work_stn_handle = handle;
for (loop = 0; loop < 10; loop++) work_in[loop] = 1;
work_in[10] = 2;
v_opnvwk(work_in,&work_stn_hndle,work_out);
}
clean_up( )
{
v_clsvwk(work_stn_hndle);
appl_exit( );
}

inch_to_meter( )
{
float inches;     /*how many inches to express in meters*/
float meters;     /*how many meters*/

printf("\n\nHow many inches? ");
scanf("%f",&inches);
meters = inches / METER_CONV;
printf("\n%8.2f inches equals %8.2f meters.\n\n",inches,meters);
}

cub_inch_to_liter( )
{
float cubic_inches;     /*how many cubic inches*/
float liters;           /*how many liters*/

printf("\n\nHow many cubic inches? ");
scanf("%f",&cubic_inches);
liters = cubic_inches / LITER_CONV;
printf("\n%8.2f cubic inches equals %8.2f liters.\n\n"
,cubic_inches,liters);
}

ounce_to_grams( )
{
float ounces;      /*how many ounces*/
float grams;       /*how many grams*/

printf("\n\nHow many ounces? ");
scanf("%f",&ounces);
grams = ounces / GRAM_CONV;
printf("\n%8.2f ounces equals %8.2f grams.\n\n",ounces, grams);
```

This program will allow you to convert inches to meters, cubic inches to liters, and ounces to grams. You may continue to make numerous conversions without having to rerun the program, and the modular nature of the conversion procedures allow you to add more options easily.

Program Explanation

We begin, of course, with our program name as a comment. The **#include** directive is then used to include the **stdio.h** and **gembind.h** header files. Next, the **#define** directive is used to set up constants for our conversions. The global variables are then declared and immediately thereafter are the blocks containing the main routine, our usual setup and cleanup routines, and three conversion routines: **inch_to_meter**, **cub_inch_to_liter**, and **ounce_to_gram**. Now let's look at each block in detail in order to determine exactly how they fit together.

The main program block begins by declaring a **done** variable to indicate when the user is finished performing conversions, a **valid** variable to indicate whether the user's input is valid, and a **choice** variable to hold the user's input. The **window_set_up graf_mouse** routines are then called to set up the program. Each of these procedures has already been explained and should warrant no further discussion, other than mentioning that it is indeed good practice to develop a series of functions that you will use at the beginning of your interactive programs so that they are consistent; otherwise your programs may look rather shabby and be difficult to debug.

Immediately after this initialization, the **done** is set to **FALSE**, and we begin the main loop with a **while (!done)** condition. The **valid** is initialized to **FALSE**, and a nested **while (!valid)** loop is started that will be directly affected by valid input from the user. A menu is then displayed that shows four options for the user to choose to continue. As shown in the successive **printf** statements, the options include each of the three conversions and the option to quit the program. A **getchar** is then executed, and whatever character the user entered at the menu is then used in a **switch** statement to determine the appropriate action. As the **switch** statement reads, if the user entered a 1, 2, or 3, then the appropriate procedure is invoked, and **valid** is set to **TRUE** because the user's response was acceptable. If a 4 was entered, both variables **valid** and **done** are set to **TRUE** because the user wishes to quit, and the response was acceptable. The **default** clause of the **switch** statement is then used to display an error message to the user, and no variables (flags) are changed.

When you now look back up at the two previous **while** loops, it should

be obvious that the outer one (**while (!done)**) terminates only when a 4 is entered, whereas the inner one (**while (!valid)**) terminates when any value from 1 to 4 is entered.

The three procedures are quite similar in that they all request an amount of some type (inches, cubic inches, or ounces) and then convert that amount into its equivalent in the metric system (meters, liters, or grams) through the use of a conversion constant (**METER_CONV**, **LITER_CONV**, or **GRAM_CONV**). You may also have noticed that each of these three procedures uses the formatted **printf** statement (for example, **%8.2f**), where the first digit (8) specifies the number of characters to be printed (field width), and the second digit (2) specifies how many digits will be displayed to the right of the decimal point.

Be sure to save this program on your disk so that you may now look at the next program, which guesses a number in the least average number of attempts.

Guess a Number Program

This particular program was designed to guess a number you would choose from 1 to 100 in the least number of tries. In fact, we guarantee that it will require no more than seven guesses to determine your number!

The secret behind the program is called a binary search, and it works as follows:

1. first, a range is set up for possible values (in this case, 1 to 100);
2. next, the user is asked to pick a number within that range;
3. the midpoint of possible values is then calculated with the formula: (maximum value + minimum value) ÷ 2, which would yield 50 ((100 + 1) ÷ 2 = 50);
4. the program then asks you if your number is a) equal to this midpoint, b) greater than this midpoint, or c) less than this midpoint;
5. if it is equal to your number, then the program stops; otherwise, if your number is greater, then the bottom half of possible values (1–50 for the first guess) are thrown out, and we make the midpoint (or guess) the lowest possible value (actually, midpoint + 1 is the lowest possible value, but due to integer truncation we can ignore the "+

1") and go back to step 3. If your number is less than the guess, then the top half of possible values (50–100 for the first guess) are thrown out, and we make the midpoint (or guess) the highest possible value and go back to step 3.

This loop from step 5 to step 3 will continue until your number has correctly been identified and, with the given range of 1 to 100, will take no more than seven attempts. Seven is not some magical number that holds true for all ranges, however; rather, this figure is derived from determining the lowest power of 2 that exceeds the range size. To understand this concept more clearly, take a moment to review the following table of powers of 2:

Power	*Expression*	*Equivalent*
0	2^0	1
1	2^1	2
2	2^2	4
3	2^3	8
4	2^4	16
5	2^5	32
6	2^6	64
7	2^7	128
8	2^8	256
9	2^9	512
10	2^{10}	1024

As you can see, the lowest power of 2 that results in a figure equal to or greater than our range size (100) is 7. Therefore, it will take at most seven guesses before the number is correctly picked. Also, the reason we raise 2 to a particular power is because we are halving the list of possibilities with each guess.

Now that you have an understanding of how the program works, enter it into your ST as it appears below:

```
/*Number Guessing Program*/

#includ (stdio.h)
#include (gembind.h)
```

```
#define MAX_GUESS 7    /*maximum guesses the program will make*/
#define MIN_START 1    /*initial min_num value*/
#define MAX_START 100  /*initial max_num value*/


int work_stn_hndle,handle;
int nothing;
char go_ahead;     /*input character from user*/
int found;         /*has your number been found?*/
int guess;         /*the actual guess*/
int done;          /*are you finished with the game?*/
int max_num;       /*the current maximum range of guesses*/
int min_num;       /*the current minimum range of guesses*/
int count;         /*the number of guesses which have been made*/
int valid;         /*did you provide a valid response?*/

main()
{
window_set_up();
graf_mouse(M_OFF,&nothing);
done = FALSE;
while (!done)
  {
  min_num = MIN_START;  /*set the range from 1 to 100*/
  max_num = MAX_START;
  printf("Pick a number between 1 and 100 . . . \n");
  printf("Press any key to continue . . . \n\n");
  go_ahead = getchar();
  found = FALSE;
  guess = (max_num + min_num)/2;
  count = 1;
  while ((!found) && (count <= MAX_GUESS))
    {
    printf("\nIs your number less than (L), greater than (G),");
    printf("or equal to (E)%-3d?",guess); /*left justify guess*/
    switch (go_ahead = getchar())
      {
      case 'E':
      case 'e': found = TRUE;
            break;
      case 'L':
      case 'l': max_num = guess;
          guess = (max_num + min_num)/2;
          ++count;
```

```
            break;
        case 'G':
        case 'g': min_num = guess;
            guess = (max_num + min_num)/2;
            ++count;
            break;
        default: printf("\nINVALID RESPONSE . . . PLEASE TRY
        AGAIN!!\n");
        }
    }
if (count <= MAX_GUESS)
    printf("\nSee, we guessed your number in only %1d
    guesses!\n",count);
else
    {
    printf("\n\nYou must have missed your number because the
    program\n");
    printf("has already made the maximum number of guesses!\n");
    }
valid = FALSE;
while (!valid)
    {
    printf("\nWould you like to play again (Y/N)?");
    switch (go_ahead = getchar())
        {
        case 'Y':
        case 'y': valid = TRUE;
                  break;
        case 'N':
        case 'n': done = TRUE;
                  valid = TRUE;
                  break;
        default: printf("\nINVALID RESPONSE . . . PLEASE TRY AGAIN!");
        }
        printf("\n\n");
        }
    }
graf_mouse(M_ON,&nothing);
clean_up();
}

window_set_up()
{
short loop;
```

```
int work_in[11];
int work_out[57];
int dummy;

appl_init( );
handle = graf_handle(&dummy,&dummy,&dummy,&dummy);
work_stn_hndle = handle;
for (loop = 0; loop < 10; loop++) work_in[loop] = 1;
work_in[10] = 2;
v_opnvwk(work_in,&work_stn_hndle,work_out);
}

clean_up( )
{
v_clsvwk(work_stn_hndle);
appl_exit( );
}
```

Program Explanation

This program contains no functions other than the main and initialization/cleanup routines and is therefore easily read from top to bottom. After our constant and variable declarations, we begin again by setting up the screen for input/output with the user.

After this screen initialization takes place, we begin our main loop (**while (!done)**), which will continue until the user decides to stop playing. We immediately assign values to our **min_num** (1) and **max_num** (100) variables, and through the use of **printf** and **getch** we ask the user to pick a number between 1 and 100 and wait for any key to be pressed to continue. We then initialize **found** to **FALSE** and use the formula for finding the midpoint between two numbers to determine our first guess (the formula simply adds our **max_num** and **min_num** figures and then divides the sum by 2 to arrive at an average, or midpoint, of the two numbers). The integer count is then initialized to 1 and through the next "while" loop (**while (!found) and (count <= MAX_GUESS)**) is not permitted to exceed 7. Rather than saying:

```
while ((!found) && (count <= 7))
```

we chose to assign a constant (**MAX_GUESS**) to represent 7 for two reasons: 1) it makes the statement more readable, and 2) it lends itself to a more obvious change should anyone choose to modify the program to allow the

range of numbers to be 1 to 1000. (This is because, from the power of 2 chart above, you would have to make the maximum number of guesses equal to 10, and you may forget what the 7 represented; but with the constant name **MAX_GUESS**, it should be more meaningful.)

So we begin this "while" loop and continue to execute it until we either find the user's number or we have made the maximum number of guesses. After starting this loop, the user is asked to say whether his/her number is less than, greater than, or equal to the calculated guess. A **switch** statement is then performed on the response and:

1. if the guess is equal to the user's number, **found** is set to **TRUE** so that the **while((!found) && (count <= MAX_GUESS))** loop will terminate;
2. if the user's number is less than the guess, the top half of possible numbers is thrown away (by assigning **max_num = guess**) and a new guess is calculated;
3. if the user's number is greater than the guess, the bottom half of possible numbers is thrown away (by assigning **min_num = guess**) and a new guess is calculated; or
4. if the user enters in an invalid response (the default clause), a message is displayed.

[Note that both uppercase and lowercase letters are used in the **switch** statement, so that the user can have the Caps Lock key down or up and the input will be valid.]

Next, when the program has either found the number or has guessed the maximum number of guesses, if the user's number was guessed (meaning that **count <= MAX_GUESS** is **TRUE**), a message is displayed that says that it took only **count** guesses to determine it, or a message is displayed that tells the user that the maximum number of guesses has been made and since the number was not found, he/she must have missed it. When either of these events is complete, the other nested "while" loop (**while (!valid)**) is executed. This loop is used to ask the user if he/she would like to play again, and based on the user's input, the flags **valid** (if a 'Y', 'y', 'N', or 'n' is entered) and **done** (if a 'N' or 'n' is entered) are set to **TRUE**. As with other inputs of this nature, if an invalid response is entered (the default clause) then an error message is displayed.

Decimal to Hexadecimal Conversion Program

The next program in this chapter may be used to convert any decimal (base 10) integer value into its hexadecimal (base 16) equivalent. If you are familiar with hexadecimal notation, simply enter the program, and follow its instructions to perform conversions. For those of you who are unfamiliar with hexadecimal numbers, or "hex" as they are commonly referred to, the following is a brief explanation of their use and significance in the computer world.

Since we were small children, we have used the base 10 numbering system; we simply use the digits 0 to 9 to represent any number (there are 10 unique digits (0, 1, . . ., 8, 9) in this system). Let's take a look at the number 425. What this number really represents is four hundreds, two tens, and five ones, because the first digit (5) is in the one's position, the second digit (2) is in the ten's position, and the third digit (4) is in the hundred's position. If we multiply and add these figures together, we get:

$$\begin{aligned} 4 \times 100 &= 400 \\ 2 \times 10 &= 20 \\ 5 \times 1 &= 5 \\ \hline & 425 \text{ TOTAL} \end{aligned}$$

Also note that since we are using base 10, each digit is multiplied by 10 raised to its position minus 1, or mathematically:

$$\text{decimal place value} = \text{digit} \times 10^{(\text{position}-1)}$$

For instance, the second decimal place's value is:

$$2 \times 10^{(2-1)} = 2 \times 10^{1} = 2 \times 10 = 20$$

as was stated above.

This discussion may seem quite trivial, but now keep in mind that the same rules apply for hexadecimal numbers, except that now we have 16 unique digits, and each hexadecimal place in the number is multiplied by a power of 16. First of all, the 16 digits for hexadecimal notation are:

Hex	*Decimal Equivalent*
$0	0
$1	1
$2	2
$3	3
$4	4
$5	5
$6	6
$7	7
$8	8
$9	9
$A	10
$B	11
$C	12
$D	13
$E	14
$F	15

As you can see, the digits 0 through 9 are equivalent in both systems, and the values 10 through 15 are represented by the first six letters in the alphabet. In addition, hex numbers are usually preceded by a dollar sign to differentiate them from a base 10 value.

Now let's look at the hex number $1A9. What is this equivalent to in base 10? If we use our rules from above, we would multiply and add to get the following:

$$\begin{aligned} \mathbf{1} \times 16^{(3-1)} &= \mathbf{1} \times 16^{2} = 256 \\ \mathbf{10} \times 16^{(2-1)} &= \mathbf{10} \times 16^{1} = 160 \\ \mathbf{9} \times 16^{(1-1)} &= \mathbf{9} \times 16^{(0)} = \quad 9 \\ & \qquad\qquad\qquad\quad 425 \text{ TOTAL} \end{aligned}$$

The hex values are represented in boldface to show how they fit into the computations; compare them to the base 10 values from our earlier example to understand the similarity figuring the total. (Also, recall that $A is equal to 10 in base 10.)

Now that you know how to translate a hex number into base 10, let's see what's involved in converting from decimal to hex; all you have to remember is to divide by 16 and write the remainder. For example, to convert 425 into hex we first divide it by 16:

$$425 \div 16 = 26 \text{ with a remainder of } \mathbf{9}$$

we then divide the resulting dividend (26) by 16:

$$26 \div 16 = 1 \text{ with a remainder of } \mathbf{10}$$

we then divide this resulting dividend (1) by 16:

$$1 \div 16 = 0 \text{ with a remainder of } \mathbf{1}$$

and, since our dividend is zero, we stop. We have determined that 1, 10, and 9 are the remainders (from last to first), and if we convert them to their respective hex equivalents (basically, only 10 must be converted to A), we find that the hex equivalent of 425 is $1A9.

You now know how to go from decimal to hex and vice versa, but you may still be wondering how this knowledge could ever be useful. In computer applications, numbers are quite often expressed in systems other than base 10, and hexadecimal notation is one of the most popular of these systems. Therefore, it is worth your while to try a few conversions on your own and check them by either converting back to the original system or by using the following program:

/*Hexadecimal Conversion Program*/

```
#include <stdio.h>
#include <gembind.h>

int work_stn_hndle,handle;

main()
{
int done;
int valid;
int number = 0;
char answer;
int nothing;
```

```
window_set_up( );
graf_mouse(M_OFF,&nothing);
done = FALSE;
while (!done)
{
printf("Please enter a decimal integer without commas or decimal
point. ");
scanf("%d",&number);
printf("The hexadecimal equivalent of %d is %x\n",number,number);
valid = FALSE;
while (!valid)
  {
  printf("Would you like to try another number (Y/N)?");
  answer = getchar( );
  switch (answer)
    {
    case 'Y':
    case 'y': printf("\n");
              valid = TRUE;
              break;
    case 'N':
    case 'n': valid = TRUE;
              done = TRUE;
              break;
        default:printf("\nINVALID RESPONSE . . . PLEASE TRY
        AGAIN!\n");
    }
    }
  }
graf_mouse(M_ON,&nothing);
clean_up( );
}

window_set_up( )
{
short loop;
int work_in[11];
int work_out[57];
int dummy;

appl_init( );
handle = graf_handle(&dummy,&dummy,&dummy,&dummy);
work_stn_hndle = handle;
for (loop = 0; loop < 10; loop++) work_in[loop] = 1;
```

```
work_in[10] = 2;
v_opnvwk(work_in,&work_stn_hndle,work_out);
}

clean_up()
{
v_clsvwk(work_stn_hndle);
appl_exit();
}
```

Program Explanation

This program is very simple in nature, thanks to C's convenient conversion abilities in **printf** statements.

After our usual initialization, the main "while" loop (**while (!done)**) is begun, and the user is requested to enter a base 10 integer, which is then converted within the second **printf** statement. By using the **%d** control string, the number is displayed as a base 10 integer. But when we use the **%x** control string, the number will be displayed as a hexadecimal number. Notice that we precede the hex value by the **$** sign to signify that it is a base 16 value.

This very simple conversion, which took place in one **printf** statement in C, might require a few dozen statements in, say, PASCAL, where you would have to read the number in and then look at it one digit at a time and convert it to a string so that the digits 10 through 15 (A through F) could be displayed. This is indeed a very nice feature of C, one that we will use again later in the book to create a program that will "dump" (or display) files in both hex and character form.

After we have displayed the converted number, another "while" loop is started (**(while (!valid)**), which asks the user if he/she would like to convert another number. The input is received via **getch**, and a **switch** statement is used to determine the appropriate action. If an invalid response is made, the default option in the **switch** is again used to display an error message. Finally, the program ends with our familiar **clean_up** routine to shut down the application.

Tape Counter Program

If you have ever made any recordings of your phonograph albums, you are familiar with the problem of trying to figure out exactly how many songs you can get on one side of a tape. We have all spent some time adding minutes and seconds to try and get the most out of a blank tape. But if you

take a few moments to use the following program, your future tapes will be a snap to make.

All you have to do is enter each song's length in minutes and seconds, and the program will keep a running total of the amount of time necessary to make the recording.

```
/*Tape Counter Program*/

    #include <stdio.h>
    #include <gembind.h>

    int work_stn_hndle,handle;
    int minutes;      /*song's number of minutes*/
    int seconds;      /*song's number of seconds*/
    int tot_minutes; /*total minutes of all songs entered*/
    int tot_seconds; /*total seconds of all songs entered*/

    main()
    {
    int nothing;

    window_set_up();
    graf_mouse(M_OFF,&nothing);
    tot_minutes = 0;
    tot_seconds = 0;
    do
      {
      printf("How many minutes is this song?");
      scanf("%d",&minutes);
      if (minutes != 0)
        {
        printf("How many seconds?");
        scanf("%d",&seconds);
        calc_and_disp();
        }
      } while (minutes != 0);
    graf_mouse(M_ON,&nothing);
    clean_up();
    }

    window_set_up()
    {
    short loop;
    int work_in[11];
```

```
int work_out[57];
int dummy;

appl_init( );
handle = graf_handle(&dummy,&dummy,&dummy,&dummy);
work_stn_hndle = handle;
for (loop = 0; loop < 10; loop++);work_in[loop] = 1;
work_in[10] = 2;
v_opnvwk(work_in,&work_stn_hndle,work_out);
}

clean_up( )
{
v_clsvwk(work_stn_hndle);
appl_exit( );
}

calc_and_disp( )
{
tot_minutes = tot_minutes + minutes + (tot_seconds + seconds)/60;
tot_seconds = (tot_seconds + seconds) % 60;
printf("\n\nTOTAL REQUIRED TIME SO FAR IS: %d MINUTES
%d SECONDS\n\n",
      tot_minutes,tot_seconds);
}
```

Program Explanation

The tape counter program has one main "do . . . while" loop, which continues until the user enters a zero value for the number of minutes in a song. This loop first requests the number of minutes, and then if minutes is not zero, the number of seconds is requested. Then, **calc_and_disp** is called to figure the total number of minutes and seconds thus far. The **tot_minutes** figure is calculated by adding the previous **tot_minutes** to the latest song's number of minutes. This amount is then added to the remainder after adding the **tot_seconds** to the latest song's seconds and dividing that sum by 60 as follows:

```
tot_minutes = tot_minutes + minutes + (tot_seconds + seconds)/
60;
```

The **tot_seconds** amount is then figured by using the remainder, or modulus, of adding the **tot_seconds** to the latest song's seconds and dividing by 60.

Again, the % operator is used to determine the remainder of dividing **X** by **Y (X % Y)**.

The cumulative totals are then displayed by **calc_and_disp**, and control returns to main, where the "do . . . while" loop continues. The program ends with the **clean_up** routine.

Glossary for Review

binary search—the search method used whereby the list of possible items is reduced by one-half with each successive guess

base 10 numbering system—the common numbering system that uses the ten digits 0 through 9 to represent all possible values

hexadecimal numbering system—the computer-oriented numbering system that uses the 16 digits 0 through F to represent all possible values

Quiz

1. Why weren't **min_num** and **max_num** (originally 1 and 100, respectively) declared as constants with the **#define** directive in the number guessing program?
2. Why weren't the three routines **inch_to_meter**, **cub_inch_to_meter**, and **ounce_to_gram** combined into one routine, since they all perform some sort of conversion?
3. Why was "do . . . while" loop used in the tape counter program instead of a regular "while" loop?
4. Are there any numbers that have the same representation in both hexadecimal and base 10 notation?

Chapter 8

Advanced Data Structures and Concepts

In this chapter, you will learn:

- More about data types.
- The concept of arrays, and how they are used in programs.
- About the **struct** and **file** types.
- The concept of recursion, and how to use it in C.

Advanced Numeric Types: Longs, Doubles, Words, and Unsigned Types

We have already used the **LONG**, **short**, and **WORD** variable types, but it is worth noting some of the reasons behind using them versus just declaring an int type. We have stated that the maximum range for a long int is from −2,147,483,648 to 2,147,483,647 and should accommodate most of your needs. You will probably never need to work with a value outside this range, and, in fact, it would be more efficient if you could use a variable that did not take up the four bytes that a long int uses. To this end, C provides the **WORD** or **short** declaration types. **WORD** and **short** are interchangeable in Lattice C in that they use only two bytes of memory (**WORD** is declared as **#define WORD short**). On other compilers, such as Megamax, **WORD** is defined in the **portab.h** file as an int (two bytes), whereas a short int is defined to be one byte. The **WORD** or int type range is from −32,768 to 32,767, which still provides a rather large range of values, but only requires half the memory of a long int.

In addition to these fundamental integer types, a variable can be declared to be an **unsigned short**, **int**, **LONG**, or **WORD**. For example, an **unsigned int** would have a range of 0 to 65,535. The **unsigned int** may hold a larger number than the regular int, but it still does so in only two bytes. How is this possible? In an int, the two bytes are comprised of 16 bits (8 bits per byte), and one of the bits is used to determine if the number is positive or negative and is commonly referred to as the *sign bit*. The **unsigned int** does not need to use one of its bits as an indicator of the sign and can therefore result in producing a larger positive number. However, both types have identically sized ranges, because there are only so many combinations of numbers that may be represented by 16 bits!

The **unsigned** modifier may be used on any of the types we have used so far with the same results as described for ints. For real numbers, the float type is four bytes in length and a long float, or double, is eight bytes long and provides a more precise real value.

You will notice that many of the GEM library routines use **WORD**s, **short**s, etc. for parameters, and you should always be careful to match your variable sizes to those of the library routines. Quite often you will find a bug with a program that arises because of a size mismatch. In addition, if you are converting programs from, say, Lattice C to Megamax C, be aware that

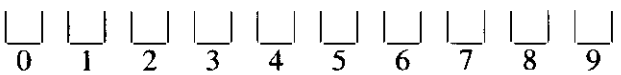
Lattice treats ints and long ints identically, and some of the parameters in the GEM functions may need to have different types in the two versions.

Simple Arrays

Up to now you have been exposed to the basic types of C including int, float, char, etc. These types are quite useful indeed, but would be somewhat limited for general programming without a mechanism to further categorize somehow groups of similar variables. The array structure is one that greatly adds to the programmer's set of tools for solving problems. We have already seen how arrays may be used to create strings like this:

```
char my_string[10];
```

This declaration reserves ten bytes of memory for storing the characters that you wish to place in **my_string**. It might be helpful to picture these ten bytes as being laid out as follows:

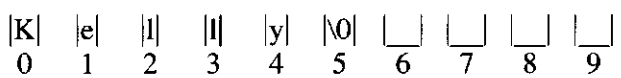


my_string

We could then assign a value to **my_string** via the **gets** function like this:

```
gets(my_string);
```

Let's say that we enter the name Kelly when the **gets** requests our input; now **my_string** looks like this:



my_string

As you can see, each of the characters in the name Kelly are stored one after another in **my_string**, and the 6th position is filled with a **\0** or a null character. This is because C has to have some way of knowing exactly how many

characters in the ten-byte **my_string** location are used to make up the string. In other words, C starts at the first location of **my_string** and continues to view each character as part of the string until a null character is encountered. Other languages have different ways of determining the length of strings. For example, many implementations of PASCAL use a byte or two before the string to hold the string length; this requires at least one extra byte per string and might therefore be viewed as excess memory usage. But remember that C needs one byte to hold the null termination character in its string representation.

Even though we declared **my_string** to hold ten characters, we could very easily assign a string with, say, 15 characters to **my_string**. What do you think would happen? If you have any variables declared after **my_string**, the contents of **my_string** will overflow into the next variable(s). Again, your variables are stored sequentially in memory, and although we specified that we only needed ten bytes of memory in our declaration of **my_string**, C looks at the declaration as an address at which to place the first character of the string; additional characters are placed one after another in memory until the end of the string. So, be careful when assigning strings, because it is very easy to trash another variable by overwriting it with a lengthy string.

Now that you are more familiar with the layout and structure of simple arrays in the form of strings, let's see how we can use the array concept to solve some programming problems. For instance, let's say you were writing a program that displays the number of wins of six baseball teams. In order to have these figures placed in variables, we would have to make declarations like this:

```
int team1;
int team2;
int team3;
int team4;
int team5;
int team6;
```

We can assign the values to these variables like this:

```
team1 = 50;
team2 = 75;
team3 = 37;
team4 = 45;
team5 = 57;
team6 = 27;
```

We could have created an array that would hold each of these values like this:

```
int team_array[6];
```

and we would make the assignments like this:

```
team_array[0] = 50;
team_array[1] = 75;
team_array[2] = 37;
team_array[3] = 45;
team_array[4] = 57;
team_array[5] = 27;
```

We say that the variable **team_array** is "an array one to six of type int." In other words, **team_array** may have up to six elements, each element is an int, and the array is indexed from 0 to 5. The last portion of that statement is worth noting: Although the index has a defined size of six, the referencing index begins at zero and goes to one less than the defined size. Picture the **team_array** as a set of six slots whose values after assignment look like this:

```
5[ 27 ]
4[ 57 ]
3[ 45 ]
2[ 37 ]
1[ 75 ]
0[ 50 ]
  team_array
```

Array elements may be assigned to one another just like other variable types. So we could say:

```
team_array[1] = 45;
team_array[2] = team_array[1];
```

which would place the value 45 in both **team_array[1]** and **team_array[2]**. Arrays don't have to contain int types; we could have an array of float, char, LONG, etc.

In general, the array's syntax is:

```
array_type var_name[number of elements];
```

where number of elements represents the range (such as [10], meaning indexed from 0 to 9), and **array_type** is the predefined type of each element.

Parallel Arrays

One of the most popular applications of arrays utilizes the concept of parallel arrays, which are arrays that are related to one another by index. Consider the situation of a teacher with 30 students who would like to write a program that prompts for each student's identification number (ID) and grade and then sorts them in descending order. The major data structures for this problem might look like this:

```
int id_array[30];
int grade_array;
```

If we were to assign the first three elements of each array based on this information:

Student ID	*Grade*
1234	84
5678	100
3456	91

with this code:

```
id_array[0] = 1234;
grade_array[0] = 84;
id_array[1] = 5678;
grade_array[1] = 100;
id_array[2] = 3456;
grade_array[2] = 91;
```

the arrays would look like this:

2[3456]	2[91]
1[5678]	1[100]
0[1234]	0[84]
id_array	grade_array

As you may have noticed in the assignment statements above, each individual student's ID and grade are located in matching locations in the two arrays (i.e., ID 5678 is in slot 1 of the **id_array**, and that student's grade is also in slot 1 of the **grade_array**). This may seen rather trivial on first inspection. However, when the items are later sorted in descending grade order, it is imperative that the relationship remain intact. Otherwise there would be no way of knowing which ID corresponds to which grade!

Structures and Files

The student ID/grade example discussed earlier might better be solved through the use of a record data structure. A *structure* or, record, is a collection of one or more different item(s) or field(s) that are grouped together for logic and convenience. For instance, this declaration would provide a structure for our student ID/grade problem:

```
struct student_rec
        {
        int id;
        int grade;
        };
```

In our **student_rec** type we say that we have two fields: **id** and **grade**. In general, the structure type declaration looks like this:

```
struct rec_type
        {
        type1 field1;
        type2 field2;
            :
            :
        typen fieldn;
        };
```

We must next declare a variable of this type so that we may work with it in a program:

```
struct student_rec a_record;
```

In order to make an assignment to a structure's fields we must specify the structure's name followed by a dot (.) and then the field name itself. Here is how we could assign values to **a_record**:

```
a_record.id = 6789;
a_record.grade = 91;
```

It's as simple as that! But if we assign new values to those fields (such as the next ID and grade), the original values are lost. A simple solution to that would be to declare an array of structures like this:

```
struct student_rec
                    {
                    int id;
                    int grade;
                    };
struct student_rec class_array[30];
```

We could now make our student ID/grade assignments like this:

```
class_array[0].id = 1234;
class_array[0].grade = 84;
class_array[1].id = 5678;
class_array[1].grade = 100;
class_array[2].id = 3456;
class_array[2].grade = 91;
```

Instead of creating an array of structures, we could have created a file that is simply a collection of records on a disk. The following is a valid file declaration:

```
FILE *my_class;
```

This declaration says that **my_class** points to a file; the contents of the file are at this point unknown, but when the **my_class** value is used in file manipulation routines, the structure format is passed as a function parameter.

We can use the **FILE** type to store our various entries on a disk for later retrieval. To understand better the relationship between files, records, and fields, think of the filing cabinet in a doctor's office. The cabinet is a file because it is full of folders on different patients. Each patient's folder is a record, and each item in that folder (i.e., name, address, height, and weight) is a field. Now let's look at a program that uses arrays and structures to create a baseball team lineup.

Using Arrays and Structures in an Application

The following program requests information for nine players on a baseball team, including name, position, batting average, and spot in the batting order. When this information is entered for each player, the array of players (structure) is scanned through to display the batting order of the team. Let's take a look at the program line by line:

```
/*Baseball Lineup Program*/

    #include <stdio.h>
    #inculde <gembind.h>
    #include <define.h>

    struct player_rec           /*record for each player*/
          {
          char name[20];        /*player's name*/
          char position[15];    /*player's position (e.g. shortstop)*/
          int average;          /*player's batting average (e.g. 300)*/
          int bat_order;        /*player's spot in batting order*/
          };

    struct player_rec team_array[9];      /*array of players*/
    int order_set[9];                     /*batting order positions*/
    int work_stn_hndle,handle;
    int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

    main()
    {
    int nothing;

    window_set_up();
    graf_mouse(M_OFF,&nothing);
    get_player_info();
    display_player_info();
```

```
graf_mouse(M_ON,&nothing);
clean_up( );
}

window_set_up( )
{
short loop;
int work_in[11];
int work_out[57];
int dummy;

appl_init( );
handle = graf_handle(&dummy,&dummy,&dummy,&dummy);
work_stn_hndle = handle;
for(loop = 0; loop < 10; loop++)work_in[loop] = 1;
work_in[10] = 2;
v_opnvwk(work_in,&work_stn_hndle,work_out);
}

clean_up( )
{
v_clsvwk(work_stn_hndle);
appl_exit( );
}

get_player_info( )
{
int counter;      /*dummy counter*/
int valid;        /*is this entry valid?*/

for (counter = 0; counter <=8; counter++)
  order_set[counter] = FALSE;  /*initialize order_set to no
  entries*/
for (counter = 0; counter <=8; counter++)
  {
  printf("What is this player's name?");
  scanf("%s",&team_array[counter].name);
  printf("What is this player's position?");
  gets(team_array[counter].position);
  printf("What is this player's batting average? (e.g. 300)");
  scanf("%d",&team_array[counter].average);
  do
  {
  valid = TRUE;
  printf("What is this player's spot in the batting order (e.g.2)?");
```

```
    scanf("%d",&team_array[counter].bat_order);
    if(order_set[team_array[counter].bat_order])
      {
      printf("\nThat spot in the order has already been used!\n\n");
      valid = FALSE;
      }
    else
      {
      order_set[team_array[counter].bat_order] = TRUE;
      printf(("\n");
      }.
      }
    }
}

display_player_info()
{
int which_batter;      /*dummy counter*/
int counter;           /*dummy counter*/

printf("\n\nYour batting order is as follows:\n\n");
for(which_batter = 1; which_batter <=9; which_batter++)

{
  counter = 0;
  while (team_array[counter].bat_order!= which_batter)
    ++counter;
  printf(" Batter %d is %s who is playing %s/n", which_batter,
        team_array[counter].name,team_array[counter].position);
  }
printf("\n\nPRESS RETURN TO CONTINUE. . .");
getchar();
}
```

When you run this baseball program, you are requested to enter player name, position, batting average, and position in the batting order for nine team members. If for any reason you should enter two players with the same batting order position, the program will note this discrepancy with an error message and request you to enter another batting order position for that player.

The baseball program has two functions—one for gathering the player information (**get_player_info()**) and another for displaying the information (**display_player_info()**).

The main program's declarations begin with a structure for the players

that contains fields for name, position, batting average, and position in the batting order. The **team_array** array is then declared so that it may hold nine player records, and an array called **order_set** is declared to hold nine elements that indicate whether each of the nine batting order positions is already occupied. The main program merely performs the usual initialization and calls both **get_player_info()** and **display_player_info()**.

The routine **get_player_info()** begins by initializing the **order_set** to all **FALSE** values. Then for each player the name, position, average, and batting order position are requested. During the request for batting order position the **order_set** is used to make sure that no two players have the same position in the batting order (which would make no logical sense, but because the user could mistakenly enter the same number twice, we have incorporated this error handling mechanism). Again, if it is determined that the number the user entered for the batting order is the index of an entry that is already set to **TRUE**, an error is displayed; otherwise, that slot in the **order_set** is set to **TRUE**.

The function **display_player_info()** simply scans through the **team_array** for each successive position in the batting order, and with each one that it finds, it performs a **printf**, which displays the player's name and position.

Enter the lineup of your favorite team, or make one up to see how the program works in various situations. Be sure to attempt to enter the same batting order position for two or more players.

As you can see, advanced data structures like arrays and records have significant use within C and in fact are complementary to one another in assisting the programmer with smooth, readable code.

Recursion (or, Can I Call Myself?)

We have seen how functions can be used in C to simplify the task of writing lengthy programs by centralizing common operations and separating logical blocks of code. We have also explained that functions can be called or referenced by the main routine or other routine depending on the program scope. One point we have not discussed, however, is that of a function calling itself. For example, let's say we want to write a program that, given a particular integer, calculates the product of itself and every integer between itself and 1 (i.e., given 4, the result is 4*3*2*1 = 24). (This mathematical concept is known as "factorials" and is written as 4!, which is read "four factorial.") This program accomplishes the job:

/*Recursion Program*/

```
#include <stdio.h>
#include <gembind.h>
#include <define.h>
int work_stn_hndle, handle;
int done;              /*are we finished yet?*/
int valid;             /*is this a valid number?*/
int fac_num;           /*number to factorialize*/
int temp_num;          /*temp variable used to perform factorials*/
long float result;     /*the result of factorializing again*/
char again;            /*Y/N answer to calculate again*/
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

main()
{
int nothing;

window_set_up();
graf_mouse(M_OFF,&nothing);
done = FALSE;
while (!done)
  {
result = 1;     /*initialize result*/
valid = FALSE;
while (!valid)
  {
  printf("Please enter an integer between 1 and 150 which you\n");
  print("want to factorialize. . .");
  scanf("%d",&fac_num);
  if ((fac_num > 0) && (fac_num <= 150))
    valid = TRUE;
  else
    printf("\n\nINVALID RESPONSE. . .PLEASE TRY AGAIN!\n\n\n");
  }
temp_num = fac_num;
calc_fact();
printf("\n\n\nThe result of factorializing %d is %g.\n",fac_num,result);
printf("Would you like to calculate another number(Y/N)?");
again = getchar();
if ((again == 'N') || (again == 'n'))
  done = TRUE;
  else
    printf("\n\n");
```

```
  }
graf_mouse(M_ON,&nothing);
clean_up();
}

window_set_up()
{
short loop;
int work_in[11];
int work_out[57];
int dummy;

appl_init();
handle = graf_handle(&dummy,&dummy,&dummy,&dummy);
work_stn_hndle = handle;
for (loop = 0; loop < 10; loop++)work_in[loop] = 1;
work_in[10] = 2;
v_opnvwk(work_in,&work_stn_hndle,work_out);
}

clean_up()
{
v_clsvwk(work_stn_hndle);
appl_exit();
}

calc_fact()
{
while(temp_num != 1)
  {
  result *= temp_num;
  —temp_num;
  calc_fact();
  }
}
```

The program accepts a number between 1 and 150 and through the recursive routine **calc_fact** calculates the factorial of the given number. **Calc_fact()** is said to be **recursive** because it calls itself (at the end of the "while" loop). It makes sense to set up the program in this manner, since we are repeatedly performing the same operation on a number. However, the same goal can be accomplished with another "while" loop in **calc_fact()**. In fact, most (if not all) recursive functions can be eliminated by coding the routine in a different manner. Try to avoid programming recursively for three major reasons:

1. it is very difficult for someone else (and sometimes even for yourself) to follow the logic of a recursive routine;
2. recursive routines are famous for causing programs to "crash" because they eat up a lot of memory; and
3. it is very easy to get into an infinite loop with a recursive routine where the "while" loop used to control the recursive call never has its condition check set to **FALSE**. This could occur because the loop counter is being incremented instead of decremented, etc.

At any rate, it is recommended that you exhaust every other programming method before embarking on recursive routines.

Glossary for Review

sign bit—the bit used in signed numbers that indicates whether the number is positive or negative

array—a collection of like items that are categorized by an index, which points to a location within the structure

record—a data structure that contains one or more item(s) or field(s) and is used to centralize their meaning and use

field—the smallest divisible portion of a record

file—a collection of records

recursion—the act of a function calling itself in a repeating fashion (one of the most popular examples is the mathematical factorial function)

Quiz

1. Describe the relationship between fields, records and files.
2. Why does it not make sense to try and put as many fields as possible in a recored declaration?
3. What was the purpose of the "for" loop in the baseball program, which successively set the items in the **order_set** to **FALSE** in the **get_player_info** function?

Chapter 9

More on Structures and Files

In this chapter, you will learn:

- About the organization of fields within structures and structures within files.
- The similarities between files and arrays of structures.
- Some of the most useful file handling library routines.
- How to sort and merge structures within a file.

Fields, Structures, and Files

Every day we deal with files of many different types. Your local phone book is one of the most obvious examples of a file. It consists of numerous entries, such as your own, that contain names, addresses, and phone numbers of local residents. Below is a portion of a sample phone book:

Craig, Joe D.	451 Emerson Rd.	555-1200
Staton, Mike	2009 Sidneywood Rd.	555-1201
Mackowick, Paul	1487 Altaview Ave	555-1202
Zimmerman, Terry	1234 Miamisburg Rd.	555-1203

The entire book is called a file, each entry is a structure, or record, and each structure has three fields: name, address, and number. The following C structure could be used to represent a phone book record:

```
struct phone_type
  {
  char name[25];
  char address[30];
  char number[9];
  };
```

If we then make the following variable declaration:

```
struct phone_type ph_var;
```

we could read in the fields of the structure like this:

```
printf("What is the person's name? ");
gets(ph_var.name);
printf("What is their address? ");
gets(ph_var.address);
printf("What is their phone number? ");
gets(ph_var.number);
```

We would then want to put these in what we have been referring to as a file but we don't know how to do this yet so let's look at another alternative to storing this information in a familiar manner.

Files vs. Arrays of Structures

If we wanted to work with several phone book entries within a program we could declare a variable like this:

```
struct phone_type ph_bk_array[10];
```

which would allow us to read in up to 10 entries like this:

```
for (i = 0; i < 10; i++)
  {
  printf("What is the person's name? ");
  gets(ph_bk_array[i].name);
  printf("What is their address? ");
  gets(ph_bk_array[i].address);
  printf("What is their phone number? ");
  gets(ph_bk_array[i].number);
  }
```

We could then manipulate this array so that it is sorted alphabetically by last name, phone number, etc. The main problem arises when we want to save the information for future use; when you end the program, the array's contents are forever lost, and although you could have declared a larger array, you will always be limited by the size of the array if you want to work with many more entries.

The C file type could be used to simplify this problem. We would need to declare a file within the program such as this:

```
FILE *phone_file;
```

This declaration will later allow us to place multiple **phone_type** structures on a disk and in a manner that we can later retrieve and manipulate. The file is then the obvious choice for storing your phone book entries, since the information may be retrieved after the program has been terminated, and the number of entries is usually limited only by the available disk space. The manipulation of files involves four major C functions: **fopen**, **fread**, **fwrite**, and **fclose**.

File Handling Library Routines

In order to access a file for reading or writing, you must first open it with this function:

```
ptr_to_file = fopen(file_name,access_type);
```

where **ptr_to_file** is the **FILE** variable used in the program to reference the file (e.g., **phone_file** as declared above), **file_name** is the name of the file as it appears on the disk (or as it will be named when viewed as an icon in the disk window), and **access_type** is the mode for which the file is to be opened. Possible values for **access_type** include "r" for read mode, "w" for write mode, and "a" for append mode.

Once you have opened a file, you may then either read or write to it using these functions:

```
fread(&struct_var,struct_size,num_of_structs,ptr_to_file);

fwrite(&struct_var,struct_size,num_of_structs,ptr_to_file);
```

where, again, **ptr_to_file** is the **FILE** variable used in the program to reference the file, **struct_var** is the item (notice that the address is passed via the **&** operator) you wish to read from or write to the file, **struct_size** is the size of the **struct_var**, and **num_of_structs** is the number of structures to be read/written. Rather than figuring exactly how large the **struct_var** is, it is common to use the **sizeof** operator like this:

```
fread(&struct_var,sizeof(struct_var),num_of_structs,ptr_to_file);
```

or:

```
fread(&struct_var,sizeof(struct
struct_type),num_of_structs,ptr_to_file);
```

Either one of these methods will provide the desired results, since the **sizeof** operator will determine the exact size of the **struct_var**, whether you use **sizeof** with the variable (**struct_var**) or its structure type (**struct_type**). Be careful, however; if you do use the structure type, make sure that you precede it with the reserved word **struct**.

When you are finished with a file, you should always close it with this function:

```
fclose(ptr_to_file);
```

There should always be a corresponding **fclose** performed upon every **fopen**-ed file. If you do not follow that simple rule, you could end up with many problems in your programs.

Finally, when reading from a file, it is an error to attempt to read beyond the last entry in the file or to try to read from a nonexistent file, since there is nothing there! You can ensure that you have successfully read in an entry (and therefore not be at the end of the file) by setting up your reading loop like this:

```
while (fread(&phone_record,sizeof(struct phone_type),
1,phone_file)==1);
```

As long as the result of **fread** is equal to 1, we have been able to read another entry, and we aren't at the end of the file.

In order to avoid the problem of reading from a nonexistent file, we can check the result placed in the file pointer against the **NULL** value (as **#defined** in the **portab.h** file in Megamax); if it equals **NULL**, the file does not exist on the disk. Here's an example of how to check for this situation:

```
if((phone_file = fopen(name,"r"))!= NULL)
  /*it's safe to attempt an fread*/
```

Using a File in a Phone Book Program

The concepts described above are brought out in this program, which may be used to keep your own private phone book on a disk:

/*Phone Book Program*/

```
#include ⟨stdio.h⟩
#include ⟨portab.h⟩
#include ⟨gembind.h⟩
#include ⟨define.h⟩

struct phone_type
  {
```

```
  char first_name[10];   /*entry's first name*/
  char last_name[10];    /*entry's last name*/
  char number[9];        /*entry's phone number*/
  };

int work_stn_hndle,handle;
int nothing;
char user_file_name[10];  /*user's phone book file name*/
FILE *phone_file;         /*phone book file pointer*/
char response;            /*menu selection*/
int done;                 /*are we finished?*/
struct phone_type phone_record; /*variable for reading/writing to file*/
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

main()
{
window_set_up();
graf_mouse(M_OFF,&nothing);
done = FALSE;
printf("Enter your file's name: ");
scanf("%s",&user_file_name);
while (!done)
  {
  printf("\n\nWhich do you wish to do:\n\n");
  printf("  1. Add a record to the file\n");
  printf("  2. Search the file\n");
  printf("  3. QUIT\n\n");
  printf("PLEASE ENTER A 1, 2 OR 3 ");
  switch(response = getchar())
    {
    case '1':add_record();
             break;
    case '2':search_file();
             break;
    case '3':done = TRUE;
             break;
    default:printf("\n\nINVALID RESPONSE. . .PLEASE TRY
    AGAIN!\n");
    }
  }
graf_mouse(M_ON,&nothing);
clean_up();
}
```

```
window_set_up()
{
short loop;
int work_in[11];
int work_out[57];
int dummy;

appl_init();
handle = graf_handle(&dummy,&dummy,&dummy,&dummy);
work_stn_hndle = handle;
for(loop = 0; loop < 10; loop++)work_in[loop] = 1;
work_in[10] = 2;
v_opnvwk(work_in,&work_stn_hndle,work_out);
}

clean_up()
{
v_clsvwk(work_stn_hndle);
appl_exit();
}

add_record()
{
printf("\n\nWhat is this person's first name? ");
scanf("%s",&phone_record.first_name);
printf("What is this person's last name? ");
scanf("%s",&phone_record.last_name);
printf("What is this person's phone number? (e.g., 555-1212) ");
scanf("%s",&phone_record.number);
phone_file = fopen(user_file_name,"a"); /*append to the file*/
fwrite(&phone_record,sizeof(struct phone_type),1,phone_file);
fclose(phone_file);
}

search_file()
{
char find_name[10];      /*name use wants to search for*/
int found;               /*have we found the name yet?*/

printf("\n\nWhat is the last name you wish to search for? ");
scanf("%s",&find_name);
if((phone_file = fopen(user_file_name,"r"))!= NULL)
  {
  found = FALSE;
```

```
    while ((!found) && (fread(&phone_record,sizeof(struct phone_type),
                        1,phone_file) == 1))
    if(strcmp(phone_record.last_name,find_name) == 0)
      found = TRUE;
    if (found)
      {
      printf("\n\n%s %s's",phone_record.first_name,
      phone_record.last_name);
      printf(" phone number is: %s\n",phone_record.number);
      }
    else
      printf("\n\nYour file contains no number for %s\n",find_name);
    fclose(phone_file);
    }
  else
    printf("\n\nThere is no file called %s on the disk!\n",user_file_name);
}
```

This program may be used to create your own disk-based phone book file, add records to it, and search for particular entries. When you run the program, you will see that a menu is displayed that asks you to select either to add a record, search the file, or quit. If you choose to add a record, the program requests the first and last name and phone number of the person to add to the file. If you choose to search the file, you are requested to enter the last name of the person you are searching; if the name is on the file, the full name and number will be displayed. After every add and search, the original menu is redisplayed so that you can perform several additions/searches without quitting the program.

The **FILE**, which is declared in the program, is called **phone_file** and is used to write/read the **phone_record** structure, which has fields for first and last names and phone number. Notice how the file pointer declaration (**FILE *phone_file;**) makes no reference whatsoever to the type of structure that is being written to that file (**phone_type**).

Aside from our usual initialization/cleanup routines, the program is broken up into the main routine and two functions, **add_record()** and **search_file()**, which place an entry at the end of the file and find a particular entry within the file, respectively.

The main program performs the initialization and asks the user for a file name (**user_file_name**) to use in the remainder of the program. The program menu is then displayed, which includes the options to add a record, search the file, and quit the program. Depending on the user's selection, either

add_record() or **search_file()** will be invoked, or the variable done will be set to **TRUE** and the program will be terminated.

When **add_record()** is invoked, the function requests the first name, last name, and phone number of the entry to be added to the file. The file is then **fopen**-ed, and the entry is appended to the end of the file, and the file is then closed via **fclose**. (Recall that the **a** option in **fopen** provides us with the ability to append to the file.)

When the **search_file()** routine is called, the user is asked which last name he/she wishes to locate in the file. The file is then **fopen**-ed, and the search is begun through a "while" loop, which continues until either the last name is found or the end of file is reached (again, we can check the end of file by comparing the result of the **fread** to 1, which would be **TRUE** if we were able to read in one record). If the name is found, the entire name is displayed along with its associated phone number; otherwise, a message is displayed that explains that the name is not in the file. In either situation, the file is then closed before returning to the main routine.

The phone book program creates a very simple file; there is no real order to the entries within the file. When a new entry is added to the file, it is placed at the end and not in alphabetical order like your local phone book. This should not create a problem for a small file that is infrequently used, but if there are quite a few entries in your file and you are searching it often, it might be nice not to have to look at almost every entry in the file before finding the one you want. If we add more logic to the program so that the file is stored in alphabetical order, you would notice a difference in speed when accessing a particular entry. To do this, we need to discuss the concepts of sorting and merging entries.

Sorting and Merging Entries

Your local phone book is a compilation of names and numbers in alphabetical order. If the book were not in any order, it would take hours to find the number of a friend. Our phone book program would have just that problem, but since the computer can search files so quickly, this lengthy period of time goes almost unnoticed, unless the file to be searched is quite large. In order to make our phone book program create a file similar to your local phone book, we must have the program sort the entries in the file before actually writing to and closing it. If we need to add an entry that would be the last one in the file, we simply add it to the end like we did in the original program.

But when we need to insert the entry at the beginning or between two existing entries, we must make room for the new entry and merge it with the existing ones. The following program has the modifications necessary to create an alphabetical phone book file:

/*Sort/Merge Phone Book Program*/

```
#include <stdio.h>
#include <portab.h>
#include <gembind.h>
#include <define.h>

struct phone_type
  {
  char first_name[10];
  char last_name[10];
  char number[9];
  };

int work_stn_hndle,handle;
int nothing;
char user_file_name[10];
FILE *phone_file;
struct phone_type phone_array[50];  /*array for sorting*/
struct phone_type phone_record;
char response;
int done;
int index;                          /*phone_array index*/
int counter;
int contrl[12], intin[128],ptsin[128],intout[128],ptsout[128];

main()
{
window_set_up();
graf_mouse(M_OFF,&nothing);
done = FALSE;
printf("Enter your file's name: ");
scanf("%s",&user_file_name);
while (!done)
  {
  printf("\n\nWhich do you wish to do:\n\n");
  printf("  1. Add a record to the file\n");
  printf("  2. Search the file\n");
  printf("  3.QUIT\n\n");
```

```
printf("PLEASE ENTER A 1, 2 OR 3. . .");
switch (response = getchar())
     {
     case '1':add_record();
              break;
     case '2':search_file();
              break;
     case '3':done = TRUE;
              break;
     default: printf("\n\nINVALID RESPONSE . . . TRY AGAIN!\n");
     }
  }
graf_mouse(M_ON,&nothing);
clean_up();
}

window_set_up()
{
short loop;
int work_in[11];
int work_out[57];
int dummy;

appl_init();
handle = graf_handle(&dummy,&dummy,&dummy,&dummy);
work_stn_hndle = handle;
for(loop = 0; loop < 10; loop++) work_in[loop] = 1;
work_in[10] = 2;
v_opnvwk(work_in,&work_stn_hndle,work_out);
}
clean_up()
{
v_clsvwk(work_stn_hndle);
appl_exit();
}

insert_and_sort()
{
int located_slot;  /*have we found the spot to insert record?*/
int dummy;         /*used to slide array elements down in sort*/

located_slot = FALSE;
counter = 0;
while ((counter < index)&&(!located_slot))
```

```
  {
  if (strcmp(phone_array[counter].last_name,
  phone_record.last_name) < 0)
    ++counter;
  else
    located_slot = TRUE;
  }
for (dummy = index; dummy > counter; dummy--)
  phone_array[dummy] = phone_array[dummy - 1];
phone_array[counter] = phone_record;
}

add_record()
{
int dummy;  /*used as index for writing to file*/

index = 0;
if((phone_file = fopen(user_file_name,"r")) != NULL)
  {
  while(fread(&phone_array[index],sizeof(struct phone_type),
            1,phone_file) == 1)
  ++index;
fclose(phone_file);
}
printf("\nWhat is this person's first name? ");
scanf("%s",&phone_record.first_name);
printf("What is this person's last name? ");
scanf("%s",&phone_record.last_name);
printf("What is this person's phone number? (e.g. 555-1212) ");
scanf("%s",&phone_record.number);
insert_and_sort();
phone_file = fopen(user_file_name,"w");
for (dummy = 0; dummy <= index; dummy++)
  fwrite(&phone_array[dummy],sizeof(struct phone_type),1,phone_file);
fclose(phone_file);
}
search_file()
{
char find_name[10];
int found;
int beyond;  /*have we gone alphabetically beyond the name?*/
beyond = FALSE;
printf("\n\nWhat is the last name you wish to search for? ");
```

```
scanf("%s",&find_name);
if ((phone_file = fopen(user_file_name,"r")) != NULL)
  {
  found = FALSE;
  while ((!found)&&(fread(&phone_record,sizeof(struct phone_type),
                          1,phone_file) == 1) && (!beyond))
    if (strcmp(phone_record.last_name,find_name) == 0)
      found = TRUE;
    else if (strcmp(phone_record.last_name,find_name) > 0)
      beyond = TRUE;  /*we've gone too far in the file*/
if(found)
  {
  printf("\n\n%s %s's",
  phone_record.first_name,phone_record.last_name);
  printf(" phone number is: %s\n",phone_record.number);
  }
else
  printf("\n\nYour file contains no number for %s\n",find_name);
  fclose(phone_file);
  }
else
  printf("\n\nThere is no file called %s on the disk!\n"user_file_name);
}
```

This new program uses a **phone_array**, which can hold up to 50 **phone_type** entries and is used to sort the file before writing it to disk. The function **add_record( )** has been changed to read the file into the array and then call the function **insert_and_sort( )**, which finds the position within the array where the insertion should be made, slides all entries below this point down one position within the array, and then places the new entry in its proper position. Finally, the function **search_file( )** no longer simply searches the file until it either finds the correct entry or hits the end of the file; it now knows that because the file is arranged alphabetically, once it gets past the point where the entry should be it quits. For example, if you are searching for the name Boyle and you are currently at the name Carlson in the file, you know that you have gone too far, and the name is not in the file.

You probably will not notice much difference in this version of the program, but to see how the file itself is completely different, use the following program to display the contents of your phone book file. Be sure to use same file name that you used in the original phone book program.

```
/*Display Phone Book Program*/

    #include <stdio.h>
    #include <portab.h>
    #include <gembind.h>
    #include <define.h>

    struct phone_type
      {
      char first_name[10];
      char last_name[10];
      char number[9];
      };
    int work_stn_hndle,handle;
    int nothing;
    char user_file_name[10];
    FILE *phone_file;
    struct phone_type phone_record;
    int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

    main()
    {
    window_set_up();
    graf_mouse(M_OFF,&nothing);
    printf("Enter your file's name: ");
    scanf("%s",&user_file_name);
    printf("\n\n");
    if((phone_file = fopen(user_file_name,"r")) != NULL)
      {
      while (fread(&phone_record,sizeof(struct phone_type),1,phone_file)
      == 1)
    printf("%s %s %s\n",phone_record.first_name,
    phone_record.last_name,phone_record.number);
      fclose(phone_file);
      }
    else
      printf("There is no file called %s on he disk!\n",user_file_name);
    printf("\n\n  PRESS RETURN TO CONTINUE. . .");
    getchar();
    graf_mouse(M_ON,&nothing);
    clean_up();
    }

    window_set_up()
```

```
{
short loop;
int work_in[11];
int work_out[57];
int dummy;

appl_init( );
handle = graf_handle(&dummy,&dummy,&dummy,&dummy);
work_stn_hndle = handle;
for (loop = 0; loop < 10; loop++) work_in[loop] = 1;
work_in[10] = 2;
v_opnvwk(work_in,&work_stn_hndle,work_out);
}

clean_up( )
{
v_clsvwk(work_stn_hndle);
appl_exit( );
}
```

Throughout our discussion of structures we have looked at variables that are declared to be of the structure type. We can also declare a variable to be a pointer to a structure type like this:

```
struct phone_type
  {
  char first_name[10];
  char last_name[10];
  char number[9];
  };

struct phone_type *an_entry;
```

We can assign the address of our **phone_record** variable to **an_entry** like this:

```
an_entry = &phone_record;
```

Then we could look at what is in **phone_record** by using the **->** operator (which works just like the dot (**.**) operator for nonpointer type structures) like this:

```
if (strcmp(an_entry->last_name,find_name) == 0)
```

which would be the same as saying this:

```
if (strcmp(phone_record.last_name,find_name) == 0)
```

One of the most important reasons for working with pointers to structures would be to pass them as parameters to functions, but for most applications, you can very easily achieve your goals by using nonpointer type structure variables.

Glossary for Review

NULL—the value that may be compared to a file pointer variable to determine if the file is on disk; if the pointer equals **NULL**, the file does not exist (found in the **portab.h** header file listed in the Megamax manual but not on the disk—so, you must create the file yourself)

sorting—the act of organizing entries in a particular order (i.e., alphabetically, numerically, etc.)

merging—the act of combining one set of entries with existing entries in a file so that the entire file is properly sorted

Quiz

1. How would you analogize a file to a collection of musical albums?
2. In the phone book example, how were we able automatically to place all of our additions to the file at the end of the file?
3. How would you modify all the phone book programs so that you don't have to put the name of your phone book file in the programs each time you run them?

Chapter 10

Debugging and File Analysis

In this chapter, you will learn:

- Some tips on debugging programs with Megamax C.
- A file dump utility that will allow you to view entire contents of files.
- A file encipher program that will protect your files from unwanted viewing.

Debugging

The program logic we have dealt with up to now has, we hope, been very easy for you to follow. Programs that are only a few dozen lines long can be written and debugged in a fairly short period of time. For those of you who are unfamiliar with the phrase *debug*, it is a term used in the computer world to represent the act of fixing program problems. A debugger is a utility that assists the programmer in fixing the problems by allowing him or her to step through the program line by line and look at or even modify variable values. So if you have run into a problem with a program, with the help of a debugger you might be able to look at a variable in question and see why it's causing you headaches.

If you are using the Lattice or Megamax C packages on your Atari ST, you probably already know that there currently is no debugging package available to help get your programs up and running. As the Lattice compiler documentation states, "A special debugger is required . . . which is not supplied with the package." Although this is an unfortunate situation, you are certainly able to debug large programs without the use of a sophisticated debugger; we prefer to say that it's just that much more challenging! And actually, without having a debugger handy, you are forced to think through the logic of your programs more than you might have to if you could rely on a debugger to look at the program while it's running. Even if you are using a C compiler with a debug package, the following few debugging tips will probably come in handy some day, and you won't have to work with a debugger to solve some problems.

First of all, you should closely monitor the activity of your program for such things as disk activity and screen display. For instance, you might notice that one string you're trying to display is making it to the screen, but the very next one is not. Or maybe you're working with a file-based program that is supposed to read/write a file on one of your disks. If you notice that the disk light never comes on before the program bombs out, you could look at the code before your disk command statements. As you can see with just these small bits of knowledge, you can better pinpoint the location of your problem.

It is also worth noting that if you accidentally say something like this:

```
if (a = b)
```

instead of this:

```
if (a == b)
```

you can wind up with some fascinating and head-scratching experiences. Some compilers may evaluate the first expression as **TRUE**, whereas others may view it as **FALSE**. At any rate, it is incorrect, and unfortunately some compilers won't even flag it as an error! So, be careful of forgetting the double equal signs when you're working with "if" statements.

If you're having problems with a particular variable and you're not sure what value it is holding, it might be a good idea to use a few **printf** statements to display the variable at strategic locations with your code. This is commonly referred to as *echo-printing* because you are echoing the value of the variable to the screen. Although this process requires an extra compilation/link or two, in the end it could save you hours of distress. In fact, the author of this book relied heavily on echo-printing to solve several strange problems! If you do use echo printing, you'll quickly discover that you should look at strings in hex rather than their string format; if for some reason the string has a null value and you try to display it, it may appear that the echo-print did not work, whereas if you were to look at the string in hex on a byte-by-byte basis, you could see that the first byte was null.

If you start working with very lengthy programs, you might find it convenient to have a debug routine that you could call at any time to look at particular variables. This is a rather awkward way to try and fix problems, but once you get the function written, you would only be bothered with inserting calls to it where you need them; you wouldn't have to worry about duplicating large blocks of code and remembering to remove them once the problems are fixed.

If you are working with a file-building program and you were able to achieve some disk activity, you could always look at what was written with the **SHOW** or **PRINT** options that are displayed when you try to open the files on the Atari desktop. Sometimes this is all you need to see exactly how much information got written to the file, but most likely you will discover that these options don't really show all the bytes that are in the file. For this reason, we have developed a file dump program that allows you to look at most of your files so that you can see exactly what they contain.

File Analysis

Let's take a look at this program:

```
/*Hex/Character file dump program*/
    #include ⟨stdio.h⟩
    #include ⟨gembind.h⟩
```

```
#define NEW_LINE '\n'
int work_stn_hndle,handle;
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

main( );
{
int nothing;
int i,j;  /*counters*/
char file_name[15];  /*file to be dumped*/
FILE *file_ptr;  /*pointer to the file*/
char a_char;  /*character in file*/
char non_hex[16];  /*the array of printable characters*/

window_set_up( );
graf_mouse(M_OFF,&nothing);
printf("Please enter the file name: ");
scanf("%s",&file_name);
printf("\n");
if ((file_ptr = fopen(file_name,"r")) != NULL)
{
for (j = 0; j < 16; j++)
  non_hex[j] = '.';  /*initialize the array*/
i = 0;
while (((a_char = getc(file_ptr)) != EOF))
  {
  if (*a_char < 16)
    printf(" 0%x",a_char);
  else
    printf(" %x",a_char);
  if ((*a_char >= ' ') && (*a_char <= '~')) /*printable*/
    non_hex[i] = a_char;
    if (i == 7) /*blank after every 8 characters*/
    printf(" ");
  else if (i == 15)
    {  /*force out the non_hex array*/
    printf(" ");
    if (non_hex[15] == NEW_LINE)
      non_hex[15] = '.';
    for (j = 0; j <= 15; ++j)
      {
     printf("%c",non_hex[j]);
     non_hex[j] = '.';  /*re-initialize array*/
     }
   printf("\n");
```

```
      i = -1;  /*reset the counter; ++i will then set it to 0*/
      }
    ++i;
    }
  if (i >= 0)  /*gracefully force out the rest of the last line*/
    {
    for (j = i; j <= 15; ++j)
      printf("  ");  /*three blanks per hex dump space*/
    if (i < 7)  /*put in extra space for break between 8 and 9*/
      printf("  ");
    printf("  ");  /*space between hex and char*/
    for (j = 0; j <= i - 1; ++j)
      printf("%c",non_hex[j]);
      }
    }
  else
    printf("NON-EXISTENT FILE ERROR!\n");
  printf("\n\n\n  PRESS RETURN TO QUIT . . .");
  getchar();
  graf_mouse(M_ON,&nothing);
  clean_up();
  }

  window_set_up()
  {
  short loop;
  int work_in[11];
  int work_out[57];
  int dummy;

  appl_int();
  handle = graf_handle(&dummy,&dummy,&dummy,&dummy);
  work_stn_hndle = handle;
  for (loop = 0; loop < 10; loop++) work_in[loop] = 1;
  work_in[10] = 2;
  v_opnvwk(work_in,&work_stn_hndle,work_out);
  }

  clean_up()
  {
  v_clsvwk(work_stn_hndle);
  appl_exit();
  }
```

The hex/character dump program is fairly straightforward and introduces only one new C function: **getc( )**. As you probably assumed, this function is used to get a character from the file specified as the parameter. We continue to call **getc( )** until the EOF value is returned.

When you run the program, you are prompted to enter the name of the file you wish to examine. Then, the file is displayed in both ASCII and the character translation. For example, if you enter the following file:

```
Hi there.
This is what a file dump looks like.
As you can see, even numbers look kind of funny!
1234567890
But, this is a very good debugging tool for files.
```

its file dump looks like this:

```
48 69 20 74 68 65 72 65   2E 0A 54 68 69 73 20 69   Hi there. . .This i
73 20 77 68 61 74 20 61   20 66 69 6C 65 20 64 75   s what a file du
6D 70 20 6C 6F 6F 6B 73   20 6C 69 6B 65 2E 0A 41   mp looks like. . .A
73 20 79 6F 75 20 63 61   6E 20 73 65 65 2C 20 65   s you can see, e
76 65 6E 20 6E 75 6D 62   65 72 73 20 6C 6F 6F 6B   ven numbers look
20 6B 69 6E 64 20 6F 66   20 66 75 6E 6E 79 21 0A   kind of funny!.
31 32 33 34 35 36 37 38   39 30 0A 42 75 74 2C 20   1234567890.But,
74 68 69 73 20 69 73 20   61 20 76 65 72 79 20 67   this is a very g
6F 6F 64 20 64 65 62 75   67 67 69 6E 67 20 74 6F   ood debugging to
6F 6C 20 66 6F 72 20 66   69 6C 65 73 2E 0A         ol for files. . .
```

Each line of the dump represents 16 bytes of the file first in ASCII and second in the converted character representation. As mentioned earlier, ASCII is simply a common method of representing characters in a form that is recognizable by computers. The hexadecimal ASCII conversion values for printable characters are as follows:

Hex Value	*Character*	*Hex Value*	*Character*
20	(space)	4F	O
21	!	50	P
22	"	51	Q
23	#	52	R
24	$	53	S

Hex Value	*Character*	*Hex Value*	*Character*
25	1/2	54	T
26	&	55	U
27	'	56	V
28	(	57	W
29	)	58	X
2A	*	59	Y
2B	+	5A	Z
2C	,	5B	[
2D	–	5C	/
2E	.	5D	]
2F	/	5E	^
30	0	5F	_
31	1	60	`
32	2	61	a
33	3	62	b
34	4	63	c
35	5	64	d
36	6	65	e
37	7	66	f
38	8	67	g
39	9	68	h
3A	:	69	i
3B	;	6A	j
3C	<	6B	k
3D	=	6C	l
3E	>	6D	m
3F	?	6E	n
40	@	6F	o
41	A	70	p
42	B	71	q
43	C	72	r
44	D	73	s
45	E	74	t

Hex Value	*Character*	*Hex Value*	*Character*
46	F	75	u
47	G	76	v
48	H	77	w
49	I	78	x
4A	J	79	y
4B	K	7A	z
4C	L	7B	{
4D	M	7C	|
4E	N	7D	}

When you analyze your files with this program you will be able to see exactly how much information got written to the file on a byte-by-byte basis until the end of the file is reached. You may have noticed that there are several dots ('.') in the character translation that actually weren't in the original file. The dots represent nonprintable characters or those that are not easily represented on the screen. These are bytes that you would not be able to see with the **SHOW** option on the Atari desktop, since that option shows you only the printable characters.

As you can see, this dump utility is quite useful in many applications. But now let's look at a program that will allow you to encipher your data files so that nobody else can see what they contain.

Try the following cipher/decipher program:

```
/*encipher/decipher program*/

    #include <stdio.h>
    #include <portab.h>
    #include <gembind.h>

    #define DISK_CIPHER "CIPHTEXT"
    int work_stn_hndle,handle;
    int nothing;
    char selection;
    short valid;
    char file_name[8];
    FILE *text_file;
    char a_char;
    FILE *cipher_file;
```

```
main()
{
window_set_up();
graf_mouse(M_OFF,&nothing);
valid = FALSE;
while (!valid)
  {
  printf("  Please select the option you would like to do:\n\n");
  printf("     1. Encipher a text file\n\n");
  printf("     2. Decipher the enciphered file\n\n");
  switch (selection = getchar())
    {
    case '1':valid = TRUE;
             encipher();
             break;
    case '2':valid = TRUE;
             decipher();
             break;
    default: printf("INVALID RESPONSE. . .PLEASE TRY
    AGAIN!\n\n");
    }
  }
printf("PRESS RETURN TO CONTINUE. . .");
getchar();
graf_mouse(M_ON,&nothing);
clean_up();
}

window_set_up()
{
short loop;
int work_in[11];
int work_out[57];
int dummy;

apl_init();
handle = graf_handle(&dummy,&dummy,&dummy,&dummy);
work_stn_hndle = handle;
for (loop = 0; loop < 10; loop++) work_in[loop] = 1;
work_in[10] = 2;
v_opnvwk(work_in,&work_stn_hndle,work_out);
}

clean_up()
```

```
{
v_clsvwk(work_stn_hndle);
appl_exit( );
}
encipher( )
{
printf("\nWhat is the name of the text file? ");
scanf("%s",&file_name);
if ((text_file = fopen(file_name,"r")) != NULL)
  {
  cipher_file = fopen(DISK_CIPHER,"w");
  while (((a_char = getc(text_file)) != EOF))
    putc((a_char + 1),cipher_file); /*add one to encipher*/
  fclose(cipher_file);
  fclose(text_file);
  printf("\n\nYOUR FILE HAS BEEN ENCIPHERED!\n\n");
  }
else
  printf("\nNO SUCH FILE ON DISK!\n\n");
}

decipher( )
{
if ((cipher_file = fopen(DISK_CIPHER,"r")) != NULL)
  {
  while (((a_char = getc(cipher_file)) != EOF))
    putchar(a_char - 1);  /*subtract one to display*/
  fclose(cipher_file);
  printf("\n");
  }
else
  printf("\nNO ENCIPHERED FILE ON DISK!\n\n");
}
```

The basic premise behind the encipher program is to add the value 1 to every byte in the file and write the results out to a file called **CIPHTEXT**. In other words, all spaces would appear as a $21, a capital A would appear as a $42, etc. Now, the **CIPHTEXT** file will hold your enciphered file, and no one will be able to read it, unless of course they know the algorithm behind the encipher program. Then, when you need to view your file, you can run the program again with the decipher option, and it will display the deciphered file on the screen. The program can be easily modified to allow you to delete

your original file, specify a user-defined name for the enciphered file, or decipher the enciphered file and write it to a file.

Glossary for Review

debugger—a utility that may allow you to view your program's execution line by line and look at and modify variables

echo printing—a debugging method of showing variable values at strategic locations within your code

Quiz

1. Why is it sometimes beneficial not to have access to a sophisticated debugger?
2. What does the following ASCII string say:

   43 20 49 53 20 46 55 4E ?

Chapter 11

Graphics

In this chapter, you will learn:

- How to work with some of the more popular graphics routines available to you in the GEM library.
- How to design programs that use a mouse-oriented routine, allowing you to utilize this important feature of the Atari ST.

Atari ST Graphics in C

As you know, your Atari ST is well-known for its graphics abilities. In fact, the level of graphics attainable on the ST is equivalent to the capabilities of systems costing four to five times as much. One of the reasons for the ST's excellent graphics capabilities is the high number of pixels, or dots, per square inch on the high-resolution monochrome monitor screen; the more dots you have compressed in a given area, the crisper the resulting picture.

In C, you also have access to the GEM graphics library, which is merely a compilation of useful graphics routines. For various reasons these routines are much faster and probably easier to use than most graphics routines you could develop yourself.

To understand better the concepts behind graphics in C, try out this program, and before linking it, be sure to use a link control file (if you are using Lattice C only), which links in the **gemlib.bin** library; just remove the asterisk before the reference to this library in the standard control file. If you are using Megamax C, you need not do anything differently, since the linker in the shell automatically pulls in the necessary libraries.

/*Graphics Example Program*/

```
#include <stdio.h>
#include <gembind.h>
#include <obdefs.h>
#include <define.h>

int work_stn_hndle,handle;
int cntrl[12], intin[128], ptsin[128], intout[128], ptsout[128];
main()
{
int nothing;

window_set_up();
graf_mouse(M_OFF,&nothing);
vsl_color(work_stn_hndle,BLACK);  /*set drawing color to black*/
vsf_interior(work_stn_hndle,HOLLOW);  /*set fill to hollow*/
draw_x();
draw_circle();
draw_empty_rec();
draw_filled_rec();
draw_filled_rec();
```

```
vst_color(work_stn_hndle,BLACK);  /*set writing color to black*/
v_gtext(work_stn_hndle,150,350,
  "PRESS RETURN TO SEE TEXT VARIATIONS. . .");
getchar( );
v_clrwk(work_stn_hndle);  /*clear the screen*/
do_text( );
v_gtext(work_stn_hndle,150,350,
  "PRESS RETURN TO QUIT THE PROGRAM. . .");
getchar( );
graf_mouse(M_ON,&nothing);
clean_up( );
}

window_set_up( )
{
short loop;
int work_in[11];
int work_out[57];
int dummy;

appl_init( );
handle = graf_handle(&dummy,&dummy,&dummy,&dummy);
work_stn_hndle = handle;
for(loop = 0; loop < 10; loop++) work_in[loop] = 1;
work_in[10] = 2;
v_opnvwk(work_in,&work_stn_hndle,work_out);
}

clean_up( )
{
v_clsvwk(work_stn_hndle);
appl_exit( );
}

draw_x( )
{
long num_of_points;   /*number of points*/
WORD point_set[4];    /*actual line end points*/

num_of_points = 2;
vsl_color(work_stn_hndle,BLACK);  /*set drawing color to black*/
point_set[0] = 50;    /*(50,50)*/
point_set[1] = 50;
point_set[2] = 125;   /*(125,150)*/
point_set[3] = 150;
```

```
v_pline(work_stn_hndle,num_of_points,point_set);  /*draw line*/
point_set[0] = 125;   /*(125,50)*/
point_set[1] = 50;
point_set[2] = 50;    /*(50,150)*/
point_set[3] = 150;
v_pline(work_stn_hndle,num_of_points,point_set);  /*draw line*/
}

draw_circle()
{

int x_coord;      /*x coordinate of circle's center*/
int y_coord;      /*y coordinate of circle's center*/
int radius;       /*circle's radius*/

x_coord = 350;  /*(350,100)*/
y_coord = 100;
radius = 50;
v_circle(work_stn_hndle,x_coord,y_coord,radius); /*draw the circle*/
}


draw_empty_rect()
{
int spec_array[4]; /*coordinate array*/

spec_array[0] = 50; /*array with (50,175), (250,325) opposite corners*/
spec_array[1] = 175;
spec_array[2] = 250;
spec_array[3] = 325;
v_bar(work_stn_hndle,spec_array);
}

draw_filled_rec()
{
int spec_array[4]

vsf_interior(work_stn_hndle,PATTERN); /*set the fill pattern to grid*/
vsf_style(work_stn_hndle,GRID);
spec_array[0] = 300; /*array with (300,175), (500,325) opposite corners*/
spec_array[1] = 175;
spec_array[2] = 500;
spec_array[3] = 325;
vr_recfl(work_stn_hndle,spec_array);
```

```
}

do_text( )
{
vst_effects(work_stn_hndle,THICKENED);
v_gtext(work_stn_hndle,50,50,"This is thickened text");
vst_effects(work_stn_hndle,SHADED);
v_gtext(work_stn_hndle,50,100,"This is shaded text");
vst_effects(work_stn_hndle,SKEWED);
v_gtext(work_stn_hndle,50,150,"This is skewed text");
vst_effects(work_stn_hndle,UNDERLINED);
v_gtext(work_stn_hndle,50,200,"This is underlined text");
vst_effects(work_stn_hndle,OUTLINE);
v_gtext(work_stn_hndle,50,250,"This is outlined text");
vst_effects(work_stn_hndle,SHADOW);
v_gtext(work_stn_hndle,50,300,"This is shadowed text");
vst_effects(work_stn_hndle,0);  /*return to no effects*/
}
```

If you run this program, you will see that it displays a screen with a large *X*, a circle, an empty rectangle, and a filled rectangle. When RETURN is pressed after this screen is displayed, another screen is displayed that shows various text formats.

Besides the usual initialization/cleanup routines, the program has five functions dedicated to drawing graphics and placing graphics text on the screen. First off, the main routine calls the routine **vsl_color** to set the line color to black; the format of this function call is:

```
vsl_color(handle,color);
```

where **handle** is the workstation handle, and **color** is any of the following values:

Color	*Value*
WHITE	0
BLACK	1
RED	2
GREEN	3
BLUE	4
CYAN	5

YELLOW	6
MAGENTA	7
LWHITE	8
LBLACK	9
LRED	10
LGREEN	11
LBLUE	12
LCYAN	13
LYELLOW	14
LMAGENTA	15

These colors are defined with the above values in the **obdefs.h** file.

Next, the interior fill style is set to hollow via the **vsf_interior** function, which has the following format:

```
vsf_interior(handle,style);
```

where **handle** is the workstation handle, and **style** is a fill style that may be any of the following:

Style	*Value*
HOLLOW	0
SOLID	1
PATTERN	2
HATCH	3
UDFILLSTYLE	4

where **UDFILLSTYLE** signifies that a user-defined fill style will be used. These constant values are also defined in the **gemlib.h** file.

After setting the interior fill style, the local routines **draw_x()**, **draw_circle()**, **draw_empty_rec()**, and **draw_filled_rec()** are called to draw the graphics screen. Then, the graphics writing color is set to black with the **vst_color** function whose format is:

```
vst_color(handle,color);
```

where **handle** is the workstation handle, and **color** is one of the values as defined above for **vsl_color**. The graphics text function, **v_gtext**, is then

invoked to display the message **PRESS RETURN TO SEE TEXT VARIATIONS . . .** ; **v-gtext** has this format:

```
v_gtext(handle,x_coord,y_coord,message);
```

where **handle** is the workstation handle, **x_coord** and **y_coord** are the coordinates where the text is to be placed, and **message** is the actual text. After a key is pressed, the screen is cleared via **v_clrwk( )**, and the local function **do_text( )** is called to display all the text formats. Then **v_gtext** is called again to inform the user that RETURN must be pressed to quite the program, and the usual cleanup routines are executed.

The **draw_x( )** function is used to draw the large *X* on the screen. There are two local variables declared in **draw_x( ): num_of_points** and **point_set**. These variables are used in the graphics library routine **v_pline** whose format is:

```
v_pline(handle,pt_count,points);
```

where **handle** is the workstation handle, and **pt_count** is the number of sets of points in the points parameter. One shortcoming with **v_pline** is that it cannot draw single points (by passing two identical end points) in its current implementation. In **draw_x( )**, **num_of_points** is set to 2 because there are two end points for each line that makes up the *X*. The routine **vsl_color** is then called to set the drawing color to black. Next, the **point_set** array is set up to hold the end points of the descending (\) line of the *X* with coordinates (50,50) and (125,150). The routine **v_pline** is then invoked to draw the line, the **point_set** is set to hold the coordinates of the ascending (/) line in the *X*, and **v_pline** is called again to draw this line.

The **draw_circle( )** function contains declarations for the center coordinates (**x_coord,y_coord**) and radius of the circle. The coordinates and radius are set to be (350,100) and 50, respectively, and **v-circle** is called to draw the circle. The **v-circle** library routine uses these parameters in this fashion:

```
v_circle(handle,x,y,radius);
```

The next two routines, **draw_empty_rect( )** and **draw_filled_rec( )**, operate in similar manners whereby an array (**spec_array**) is set up to contain the coordinates of the opposite corners of the rectangles, and the appropriate GEM routine is called to draw the rectangles. In the case of

draw_empty_rect( ), the opposite corners are (50,175) and (250,325). The routine **v_bar** is called to draw the empty rectangle with the following format:

```
v_bar(handle,coord_array);
```

For the filled rectangle in **draw_filled_rec( )**, the interior style must first be set up via **vsf_interior**, where the **PATTERN** parameter specifies that a pattern will be designated in the call to **vsf_style**. When **vsf_style** is then called, the **GRID** parameter informs GEM that we wish to fill with a **GRID** pattern. The **spec_array** is then filled to hold the opposite corners (300,175) and (500,325), and **vr_recfl** is called to draw the rectangle. So, the sequence of events to draw a filled rectangle is first to set up the pattern with both **vsf_interior** and **vsf_style**, set the opposite corners, and finally invoke **vr_recfl** with this format:

```
vr_recfl(handle,coord_array);
```

Finally, the **do_text( )** routine displays all the graphics text variations through the use of **vst_gtext** (described above) and **vst_effects**. The style is set in **vst_effects** like this:

```
vst_effects(handle,style);
```

where **style** may be one of the following values as defined in the **gemlib.h** file:

Style	*Value*
THICKENED	0x0001
SHADED	0x0002
SKEWED	0x0004
UNDERLINED	0x0008
OUTLINE	0x0010
SHADOW	0x0020

By passing a zero to **vst_effects**, all the special effects are turned off (as shown at the end of **do_text( )**). [NOTE: In the case of these and all other **#defined** constants, some compilers do not have header files with these values defined, whereas others do; depending on which compiler you are using,

you may either have to **define** them in your source code file or create a header file with them in it.]

These are only some of the graphics functions available to you in the GEM library, but as you can see, quite a few interesting graphics applications may be developed using the routines described above. Let's now look at a nifty program that uses **v_pline** and a mouse-locating routine that allows you to draw lines on the ST in high resolution.

Fun with the Mouse

The Atari ST's mouse can be used in several different ways in C. Here we will present one useful routine that allows you to know the status of the mouse's buttons and the position of the cursor on the screen.

If you enjoy drawing on your ST (with either a TV or color monitor), you'll appreciate the following mouse-based program, which allows you to draw lines in high resolution. When you run this program, press the left mouse button and release it where you want one end point, and move and click it again where you want the other end point of a line (remember that **v_pline** cannot draw single points, so if you click twice on the same point, nothing will be drawn):

```
/*Mouse Drawing Program*/

    #include <stdio.h>
    #include <portab.h>
    #include <gembind.h>

    int work_stn_hndle,handle;
    int conrtl[12], intin[128], ptsin[128], intout[128], ptsout[128];
    main()
    {
    int nothing;

    window_set_up();
    graf_mouse(M_ON,&nothing);
    graf_mouse(THIN_CROSS,&nothing);/*make mouse image thin
    cross hair*/
    mouse_draw();
    clean_up();
    }

    window_set_up()
    {
```

```
short loop;
int work_in[11];
int work_out[57];
int dummy;

appl_init( );
handle = graf_handle(&dummy,&dummy,&dummy,&dummy);
work_stn_hndle = handle;
for(loop = 0; loop < 10; loop++)work_in[loop] = 1;
work_in[10] = 2;
v_opnvwk(work_in,&work_stn_hndle,work_out);
}

clean_up( )
{
v_clsvwk(work_stn_hndle);
appl_exit( );
}

mouse_draw( )
{
int done;              /*are we finished yet?*/
int point_array[4];   /*hold points for line*/
int which_pr;        /*which button was pressed?*/
int x_pos,y_pos;   /*mouse position*/
int was_hit;         /*flag to tell if button was hit*/

done = FALSE;
vsl_color(work_stn_hndle,BLACK);  /*set drawing color to black*/
do
  {
  was_hit = FALSE;
  do
    {
    if (!was_hit)
      {
      do
        vq_mouse(work_stn_hndle,&which_pr,&x_pos,&y_pos);
      while (which_pr == 0);
      was_hit = TRUE;
      }
    vq_mouse(work_stn_hndle,&which_pr,&x_pos,&y_pos);
    } while (which_pr!= 0)
  if ((x_pos == 0) && (y_pos == 0)) /*quit if pressed at origin*/
    done = FALSE;
```

```
    else /*draw line*/
      {
      point_array[0] = x_pos;
      point_array[1] = y_pos;
      was_hit = FALSE;
      do
        {
        if (!was_hit)
          {
          do
            vq_mouse(work_stn_hndle,&which_pr,&x_pos,&y_pos);
          while (which_pr == 0);
          was_hit = TRUE;
          }
        vq_mouse(work_stn_hndle,&which_pr,&x_pos,&y_pos);
        } while (which_pr != 0);
      point_array[2] = x_pos;
      point_array[3] = y_pos;
      v_pline(work_stn_hndle,2,point_array);
      }
  }
}
```

The **mouse_draw( )** function is the routine that controls where the lines will be drawn on the screen. It uses another GEM library function, **vq_mouse**, whose format is:

```
vq_mouse(handle,&which_button,&x,&y);
```

where **which_button** represents which button was pressed. The value returned in **which_button** is 0 if no button was pressed, 1 if the left button was pressed, or 2 if the right button was pressed. The values **x** and **y** represent the coordinates of the mouse when it was polled. The concept of polling may be new to you, but it is very simple. Every time the **vq_mouse** routine is invoked, it is not looking for any particular state (e.g., button down), rather it is simply returning the position of the mouse and reporting whether either button was pressed. The first incidence of the **vq_mouse** routine is involved in a polling loop. The "do . . . while" loop keeps calling **vq_mouse** until **which_pr** is not equal to zero. In other words, the program remains in an infinite loop until a button is pressed on the mouse. For this reason, we say that the mouse buttons are being *polled* and will continue to be polled until a button is pressed.

The **vq_mouse** routine executes very quickly, and because of this, there has to be a check in the program to determine when the mouse button is released after it has been pressed. Otherwise, the possibility exists that before the button is released the first time, the next call to **vq_mouse** may have already occurred and the same point would be returned (recall that **v_pline** will not draw single points!). So, the next call to **vq_mouse** will determine if the mouse button has been released yet. If it has not been released, the next outer "do . . . while" loop will continue to execute until it is released. Notice that the inner "do . . . while" loop will not be executed at this point because the **was_hit** variable was set to **TRUE** after the button was pressed. At this point, the position of the first button press is analyzed, and if the coordinates are the origin of the screen (position (0,0), the upper left-hand corner of the screen), **done** is set to **TRUE** so that the program will be stopped. So when you want to stop the program, just click in the upper left-hand corner of the screen when you would normally click for the first point of the line.

If the first click is somewhere other than the origin, the mouse position will be saved off in the first two elements of the **point_array,** and a nested set of "do . . . while" loops identical to the set described above are executed. After these loops return the second click position, the **x** and **y** coordinates are placed in the last two positions of the **point_array**, and **v_pline** is called to draw the line.

There are many other GEM graphics routines available to you that would require an entire book to explain fully. They are fully documented in the GEM Programmer's Reference Guide from Abacus Software. At this point, you should be able to create your own drawings within C by applying all the graphics concepts discussed here.

Glossary for Review

polling—the act of repeatedly checking a device to see if any action is required within the program to reflect the status returned from the device.

Quiz

1. What would happen in the graphics example program if the last call to **vst_effects** in the **do_text( )** function is deleted?
2. What happens if the mouse is clicked at the origin for the second point in the line in the mouse drawing program?

Chapter 12

A Few Programs for the Road

In this chapter, you will learn:

- A very useful electronic calendar program that can be modified quite easily by incorporating additional routines.
- A program that may be used to calculate and maintain baseball batting averages.
- A data base program that may be used to catalog your album collection electronically.
- The concept of modularity for designing menu-driven programs.

The Date Minder Program

The first program we'd like to present to you in this chapter is one that we hope you will find helpful all year round. It's an electronic calendar that will tell you what events are coming up in the next 30 days. All you have to do is enter the information for your household (birthdays, anniversaries, etc.) so that when you need to see what's on the horizon for the next month, you simply select the "View the next month of events" option from the menu. All your calendar information will be stored in a file on your disk so that you can add more dates to the file at any time.

Let's look at the program and discuss it in detail:

```
/*Date Minder Program*/

    #include <stdio.h>
    #include <gembind.h>

    #define NAME_OF_FILE "DATES"

    struct event_rec          /*record for each event*/
      {
      int month;              /*event month*/
      int date;               /*event date*/
      char occasion[20];      /*event name*/
      };

    int work_stn_hndle,handle;
    int days_in_month[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
    charmonths[15][12] = { "January","February","March","April","May",
                                    "June","July","August","September",
                                    "October","November","December"};
    FILE *event_file;   /*file of events*/
    struct event_rec event_var_rec;   /*reads/writes event_recs*/
    int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

    main()
    {
    int nothing;

    window_set_up()
    graf_mouse(M_OFF,&nothing);
    display_menu();
    graf_mouse(M_ON,&nothing);
```

```
clean_up( );
}

window_set_up( )
{
short loop;
int work_in[11];
int work_out[57];
int dummy;

appl_init( );
handle = graf_handle(&dummy,&dummy,&dummy,&dummy);
work_stn_hndle = handle;
for (loop = 0; loop < 10; loop++) work_in[loop] = 1;
work_in[10] = 2;
v_opnvwk(work_in,&work_stn_hndle,work_out);
}

clean_up( )
{
v_clsvwk(work_stn_hndle);
appl_exit( );
}

add_dates( )
{
int valid;        /*is user's input valid?*/
int finished;     /*is user finished adding dates?*/
char response;  /*user's Y/N response for more additions*/

event_file = fopen(NAME_OF_FILE,"a");
finsished = FALSE;
while (!finished)
    {
    valid = FALSE;
    while (!valid)
      {
      printf("\n\nWhat month is this occasion(1-12)? ");
      scanf("%d",&event_var_rec.month);
      if ((event_var_rec.month > 0) && (event_var_rec.month < 13))
        valid = TRUE;
      }
      valid = FALSE;
      while (!valid)
        {
```

```
            printf("What date is this occasion (1-31)? ");
            scanf("%d",&event_var_rec.date);
            if ((event_var_rec.date > 0) && (event_var_rec.date <=
              days_in_month[event_var_rec.month - 1]))
            {
            event_var_rec.date += 40; /*avoid control characters*/
            valid = TRUE;
            }
          }
        printf("What is the occasion?\n");
        gets(event_var_rec.occasion);
        fwrite(&event_var_rec,sizeof(event_var_rec),1,event_file);
        printf("\n\nDo you have any more entries(Y/N)? ");
        response = getche();
        if ((response == 'N')||(response == 'n'))
        finished = TRUE;
        }
fclose(event_file);
printf("\n\n");
}

view_dates()
{
int this_month;  /*the month entered by the user*/
in this_date;    /*the date entered by the user*/
int next_month; /*next month (for Jan. after Dec.)*/
int valid;

valid = FALSE;
while (!valid)
  {
  printf("\n\nWhat is this month (1-12)? ");
  scanf("%d",&this_month);
  if ((this_month > 0) && (this_month < 13))
    valid = TRUE;
  }
valid = FALSE;
while (!valid)
  }
  printf(What is today's date (1-31)? ");
  scanf("%d",&this_date);
  if ((this_date > 0) && (this_date <= days_in_month[this_month -
1]))
    valid = TRUE;
```

```
    }
  printf("\n*****************************************");
  printf("*****************************************\n");
  if ((event_file = fopen(NAME_OF_FILE,"r")) != NULL)
    {
    if (this_month == 12)
      next_month = 1;
    else
      next_month = this_month + 1;
    while(fread(&event_var_rec,sizeof(event_var_rec),1,event_file) == 1)
      {
      event_var_rec.date -=40;  /*avoid control characters*/
      if (((event_var_rec.month == this_mont) &&
        (event_var_rec.date >= this_date))||
        ((event_var_rec.month == next_month) &&
        (event_var_rec.date < this_date)))
      printf("  %s is on %s %d\n",event_var_rec.occasion,
              months[event_var_rec.month - 1],
              event_var_rec.date);
      }
    printf("\n*****************************************");
    printf("*****************************************\n");
    fclose(event_file);
    }
  else
    printf("\n\n THERE IS NO DATES FILE ON YOUR DISK!\n");
  printf("\n\nPRESS RETURN TO CONTINUE . . .");
  getchar( );
  printf("\n\n");
  }

  display_menu( )
  {
  char selection;
  int valid;
  int done;

  done = FALSE;
  while (!done)
  {
  valid = FALSE;
  while (!valid)
    {
    printf("Which would you like to do:\n\n");
```

```
        printf("   1. Add dates to the file\n");
        printf("   2. View the next month of events\n");
        printf("   3. QUIT\n\n");
        printf("Enter 1, 2, or 3. . .");
        switch (selection = getchar())
          {
          case '1':add_dates();
                   valid = TRUE;
                   break;
          case '2':view_dates();
                   valid = TRUE;
                   break;
          case '3':valid = TRUE;
                   done = TRUE;
                   break;
            default:printf("\n\n\n");
            }
          }
      }
}
```

The program has a structure type called **event_rec**, which is used to hold the month, date, and name of each occasion. There are also two arrays that show how to initialize an array in the declaration; all you need to do is place all the initial values in order within a set of braces and precede the block with an equal sign. In addition, the months array is a doubly dimensioned array, which means that there are two indices. The first index of months refers to the character position of each entry, whereas the second index refers to the number of string entries. In short, the array may hold up to 12 elements of up to 15 characters each. And again, this array is initialized by placing all the initial values in order (since they are strings, they must be placed within quotes) within a set of braces.

The main routine simply performs initialization, calls **display_menu()**, and performs the cleanup. The routine **display_menu()** shows the main menu and, depending on the user's selection, either calls **add_dates()**, **view_dates()**, or quits the program.

The **add_dates()** routine is the one that actually writes the event records out to the file. The file where these dates are stored is called **DATES**. This name could be changed by simply changing the **#define** directive at the beginning of the program. At the start of **add_dates()**, the file is opened with the append option. Then, while the user is not finished adding records/events,

two loops are performed to validate the input for the month and date of the occasion. The occasion name is then entered, and the record is written to the file. The user is then asked if he/she has any more entries. If there are more, the **while (!finished)** loop is continued; otherwise the file is closed, and control is passed back to the **display_menu( )** routine.

The last function in our date minder program is called **view_dates( )**. First of all, the current date must be entered by the user. The event file is then opened, and we determine the next month using this "if" statement:

```
if (this_month == 12) then
  next_month = 1;
else
  next_month := this_month + 1;
```

As you can see, if this month is December, we want to make next month January. Otherwise, we just need to add 1 to the current month.

Then, the events that are within the next month are displayed on the screen. A very long "if" statement is used to determine whether a month fits into this category. It may be helpful to split this statement into two separate statements, either of which may be true for the date and event to be displayed:

```
1) if ((event_var_rec.month == this_month)&&
     (event_var_rec.date >= this_date))
```

or:

```
2) ((event_var_rec.month == next_month)&&
     (event_var_rec.date < this_date))
```

An example of the first case would be where today is July 10th and the event takes place on July 31st; both segments of the case are true, since the event's month (**event_var_rec.month**) is the same as **this_month**, and the event's date (**event_var_rec.date**) is greater than **this_date**.

An example of the second case would be where, again, today is July 10th and the event is on August 5th. Both segments are true here also, since the event's month (**event_var_rec.month**) is the same as **next_month** and the event's date (**event_var_rec.date**) is less than **this_date**.

When the end of the file is reached, the file is closed, and control is returned to the **display_menu( )** routine.

One other point worth noting in the date minder program is where 40

is added to the date before writing it to disk, and then it is subtracted immediately after reading it in. The reason for this is so that control characters, or characters that would not be interpreted as the number we intend them to be, would be written to the disk and may not be correctly read in.

Now let's take a look at a program that will allow you to maintain a file of your favorite baseball player's batting average.

Batting Average Program

The following program allows you to enter up to 20 baseball players, their number, number of at bats, and number of hits, and it will calculate their batting averages and store the file on disk:

```
/*Baseball Averages Program*/

    #include ⟨stdio.h⟩
    #include ⟨gembind.h⟩

    #define TEAM "BALLCLUB"

    struct player_rec

      {
      char number[4];       /*player's number*/
      char last_name[15];   /*player's last name*/
      char first_name[10]   /*player's first name*/
      float at_bats;        /*number of at bats*/
      float hits;           /*number of hits*/
      float batting_ave;    /*batting average*/
      };

    int work_stn_hndle,handle;
    FILE *bb_team_file;   /*the team's file*/
    struct player_rec a_player;       /*a player variable*/
    struct player_rec arr_of_pl[20];  /*array of players*/
    int num_of_players;               /*current number of players*/
    int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

    main()
    {
    int nothing;
```

```
window_set_up( );
graf_mouse(M_OFF,&nothing);
disp_menu( );
graf_mouse(M_ON,&nothing);
clean_up( );
}

window_set_up( )
{
short loop;
int work_in[11];
int work_out[57];
int dummy;

appl_init( );
handle = graf_handle(&dummy,&dummy,&dummy,&dummy);
work_stn_hndle = handle;
for (loop = 0; loop < 10; loop++) work_in[loop] = 1;
work_in[10] = 2;
v_opnvwk(work_in,&work_stn_hndle,work_out);
}

clean_up( )
{
v_clsvwk(work_stn_hndle);
appl_exit( );
}

disp_menu( )
{
int choice;  /*user's menu selection*/
int done;
done = FALSE;
while (!done)
  {
  printf("\n\n\nWhich of the following do you wish to
  perform:\n\n");
  printf("  1. Add a player to the file\n");
  printf("  2. Delete a player from the file\n");
  printf("  3. Enter information for a player\n");
  printf("  4. Display the file by batting average\n");
  printf("  5. QUIT\n\n");
  printf(" Please enter your selection: ");
  switch (choice = getchar( ))
```

```
        {
        case '1':add_plyr();
                 break;
        case '2':del_plyr();
                 break;
        case '3':ent_info();
                 break;
        case '4':do_disp();
                 break;
        case '5':done = TRUE;
                 break;
        default:printf("\n\nINVALID SELECTION. . .TRY AGAIN!\n");
        }
    }
}

ent_info()
{
int i;
char response;
float data;

load_array();
for (i = 0; i < num_of_plyrs; i++)
  {
  printf("\n\nDo you have any data for %s (Y/N)* ",
  arr_of_pl[i].last_name);
  response = getchar();
  if ((response == 'Y')||(response == 'y'))
    {
    printf("\nHow many additional at bats? ");
    scanf("%f",%data);
    arr_of_pl[i].at_bats += data;
    printf("How many additional hits? ");
    scanf("%f",&data);
    arr_of_pl[i].hits += data;
    if (arr_of_pl[i].at_bats != 0.0)
      arr_of_pl[i].batting_ave = arr_of_pl[i].hits / arr_of_pl[i].at_bats;
    else
      arr_of_pl[i].batting_ave = 0.0;
    }
  }
re_do_file();
}
```

```
del_plyr()
{
char 1_name[15];  /*last name of player to be deleted*/
int i,j;
int found;

printf("\n\nWhat is the last name of the player to be deleted? ");
scanf("%s",&1_name);
load_array();
i = 0;
found = FALSE;
while ((i < num_of_plyrs) && (!found))
  if (strcmp(arr_of_pl[i].last_name,1_name) == 0)
    found = TRUE;
  else
    ++i;
if (!found)
  printf("\n\nTHAT NAME IS NOT IN THE FILE!!\n");
else
  {
  for (j = i; j < num_of_plyrs - 1; j++)
    arr_of_pl[j] = arr_of_pl[j + 1];
  --num_of_plyrs;
  re_do_file();
  printf("\n\nTHAT PLAYER HAS BEEN DELETED.\n");
  }
printf("PRESS RETURN TO CONTINUE. . .");
getchar();
}

do_disp()
{
int i;

sort();
if ((bb_team_file = fopen(TEAM,"r")) != NULL)
  {
  printf("\n\n\nNUMBER NAME  AT BATS  HITS  ");
  printf("BATTING AVERAGE\n");
  printf("-----------------------------------------");
  printf("-----------------------------------------\n");
  for (i = 0; i < num_of_plyrs; i++)
    {
    printf("  %s  %-10s %15s  ",arr_of_pl[i].number,
                  arr_of_pl[i].first_name,arr_of_pl[i].last_name);
```

```
      printf("%4.0f   %4.0f   .%5.3f\n",arr_of_pl[i].at_bats,
                        arr_of_pl[i].hits,arr_of_pl[i].batting_ave);
      }
    fclose(bb_team_file);
    }
  else
    printf("\n\nTHERE IS NO BASEBALL FILE ON THIS DISK!!\n");
  printf("\n\nPRESS RETURN TO CONTINUE. . .");
  getchar( );
  }

  add_plyr( )
  {
  load_array( );
  ++num_of_plyrs;
  printf("\n\nWhat is the player's first name? ");
  scanf("%s",%arr_of_pl[num_of_plyrs - 1].first_name);
  printf("What is the player's last name? ");
  scanf("%s",&arr_of_pl[num_of_plyrs - 1].last_name);
  printf("What is his number? ");
  scanf("%s",&arr_of_pl[num_of_plyrs - 1].number);
  printf("How many at bats does he currently have? ");
  scanf("%f",&arr_of_pl[num_of_plyrs - 1].at_bats);
  printf("How many hits does he currently have? ");
  scanf("%f",&arr_of_pl[num_of_plyrs - 1].hits);
  if (arr_of_pl[num_of_plyrs - 1].at_bats != 0.0)
    arr_of_pl[num_of_plyrs - 1].batting_ave =
      arr_of_pl[num_of_plyrs - 1].hits/arr_of_pl[num_of_plyrs -
      1].at_bats;
  else
    arr_of_pl[num_of_plyrs - 1].batting_ave = 0.0;
  re_do_file( );
  }

  re_do_file( )
  {
  int i;

  bb_team_file = fopen(TEAM,"w");
  for (i = 0; i < num_of_plyrs; i++)
    fwrite(&arr_of_pl[i],sizeof(struct player_rec),1,bb_team_file);
  fclose(bb_team_file);
  }

  sort( )
```

```
{
int h,j;
struct player_rec temp_plyr;   /*temporary storage for sort()*/

load_array();
for (j = 0; j < num_of_plyrs; j++)
  for(h = 0; j < num_of_plyrs - j; h++)
    if (arr_of_pl[h].batting_ave < arr_of_pl[h + 1].batting_ave)
      {
      temp_plyr = arr_of_pl[h];
      arr_of_pl[h] = arr_of_pl[h + 1];
      arr_of_pl[h + 1] = temp_plyr;
      }
}
load_array()
{
num_of_plyrs = 0;
if ((bb_team_file = fopen(TEAM,"r")) != NULL)
  {
  while (fread(&arr_of_pl[num_of_plyrs],sizeof(struct player_rec),
         1,bb_team_file) == 1)
    ++num_of_plyrs;
  fclose(bb_team_file);
  }
}
```

The baseball averages program introduces no new concepts and should therefore be very easy for you to understand. Basically, the menu displays four options: add a player, delete a player, enter information for existing players, or display the file by order of batting average. The program contains a separate function for each of these options, as well as three other routines that are used by some or all of the menu option routines. These additional functions, **re_do_file()**, **sort()**, and **load_array()**, are responsible for re-writing the file, sorting the file in the memory array, and reading the file into the memory array, respectively. Step through this program and try entering a few statistics to verify that the program works as you expect.

Record Album Data Base Program

The final program we will show you is one that allows you to catalog your album collection electronically. Once you have entered all your albums, you

can then go back and either sequentially view the file or search the file based on album title, artist, year of release, or an extra field for your own use. Again, there are no new concepts introduced in this program, so you should have no problem following the logic:

```
/*Album Data Base Program*/

    #include <stdio.h>
    #include <gembind.h>

    #define ALB_FILE "ALBUMS"

    struct album_type
      {
      char title[25];      /*album title*/
      char artist[20];     /*album artist*/
      char year[5];/*album's year of release*/
      char xtra_field[20];  /*extra field for your own use*/
      };

    int work_stn_hndle,handle;
    struct album_type an_album;
    int done;
    FILE *album_file;
    int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

    main()
    {
    int nothing;
    char choice;

    window_set_up();
    graf_mouse(M_OFF,&nothing);
    done = FALSE;
    while (!done)
      {
      v_clrwk(work_stn_hndle);
      v_curhome(work_stn_hndle);
      printf("Select one of the following options:\n\n\n");
      printf("  1. Add albums to the file\n");
      printf("  2. Sequentially display the file\n");
      printf("  3. Search the file\n");
      printf("  4. QUIT\n\n\n");
      printf("What is your choice? ");
      switch (choice = getchar())
```

```
        {
        case '1':add_albums();
                 break;
        case '2':display_file();
                 break;
        case '3':search();
                 break;
        case '4':done = TRUE;
                 break;
        default:  /*don't do anything, screen will be erased*/;
        }
    }
graf_mouse(M_ON,&nothing);
clean_up();
}

window_set_up()
{
short loop;
int work_in[11];
int work_out[57];
int dummy;

appl_init();
handle = graf_handle(&dummy,&dummy,&dummy,&dummy);
work_stn_hndle = handle;
for (loop = 0; loop < 10; loop++) work_in[loop] = 1;
work_in[10] = 2;
v_opnvwk(work_in,&work_stn_hndle,work_out);
}

clean_up()
{
v_clsvwk(work_stn_hndle);
appl_exit();
}

search()
{
int done;
char selection;

donw = FALSE;
while (!done)
  {
```

```
    v_clrwk(work_stn_hndle);
    v_curhome(work_stn_hndle);
    printf("Which field would you like to search?\n\n\n");
  printf("   1. Group/Artist Name\n");
  printf("   2. Album Title\n");
  printf("   3. Year of Release\n");
  printf("   4. Extra field\n");
  printf("   5. QUIT\n\n\n");
  printf("Please enter your selection: ");
  switch (selection = getchar())
    {
    case '1':grp_search();
             break;
    case '2':title_search();
             break;
    case '3':year_search();
             break;
    case '4':tag1_search();
             break;
    case '5':done = TRUE;
             break;
    default:/*nothing, screen will be re-displayed*/
    }
    }
}

add_albums()
{
int done;
char *stop = "$";

done = FALSE;
album_file = fopen(ALB_FILE,"a");
while (!done)
    {
    v_clrwk(work_stn_hndle);
    v_curhome(work_stn_hndle);
    printf("What is the album's titls ($ to end)? ");
    gets(an_album.title);
    if (strcmp(an_album.title,stop) == 0) /*enter a dollar sign to end*/
      done = TRUE;
    else
      {
      printf("Who is the artist? "):
```

```
        gets(an_album.artist);
        printf("What year was the album made? ");
        gets(an_album.year);
        printf("What do you want in the extra field? ");
        gets(an_album.xtra_field);
        fwrite(&an_album,sizeof(struct album_type),1,album_file);
        }
    }
fclose(album_file);
}

display_file()
{
if ((album_file = fopen(ALB_FILE,"r")) != NULL)
  while(fread(&an_album,sizeof(structalbum_type),1,album_file)==1)
    disp_the_record();
else
  {
  printf("\n\nTHERE IS NO ALBUM FILE ON THE DISK!\n\n\n");
  printf("PRESS RETURN TO CONTINUE. . .");
  getchar();
  }
}

disp_the_record()
{
v_clrwk(work_stn_hndle);
v_curhome(work_stn_hndle);
printf("The album title is:          %s\n",an_album.title);
printf("The album artist is:         %s\n",an_album.artist);
printf("The album's release was in   %s\n",an_album.year);
printf("The extra field is:          %s\n\n\n\n",
                                     an_album.xtra_field);
printf("PRESS RETURN TO CONTINUE. . .");
getchar();
}

grp_search()
{
int found_one;     /*have we found a match yet?*/
char group[20];    /*holds name of group we wish to search*/

v_clrwk(work_stn_hndle);
v_curhome(wor_stn_hndle);
```

```
printf("Which Group/Artist do you wish to search? ");
gets(group);
if ((album_file = fopen(ALB_FILE,"r")) != NULL)
  {
  found_one = FALSE;
  while (fread(&an_album,sizeof(struct album_type),1,album_file)==1)
    if (strcmp(an_album.artist,group) == 0)
      {
      found_one = TRUE;
      disp_the_record();
      }
if (!found_one)
  {
  printf("\n\nTHAT ARTIST IS NOT IN THE FILE!\n\n");
  printf("PRESS RETURN TO CONTINUE . . .");
    getchar();
    }
  fclose(album_file);
  }
else
  {
  printf("\n\nTHERE IS NO ALBUM FILE ON THE DISK!\n\n\n");
  printf("PRESS RETURN TO CONTINUE. . .");
  getchar();
  }
}

tag1_search()
{
int found_one;
char tag1[20];

v_clrwk(work_stn_hndle);
v_curhome(work_stn_hndle);
printf("What is the extra field you wish to search? ");
gets(tag1);
if ((album_file = fopen(ALB_FILE,"r")) != NULL)
  {
  found_one = FALSE;
  while (fread(&an_album,sizeof(struct album_type),1,album_file)==1)
    if (strcmp(an_album.xtra_field,tag1) == 0)
      {
      found_one = TRUE;
      disp_the_record();
```

```
        }
    if (!found_one)
      {
      printf("\n\nTHAT EXTRA FIELD IS NOT IN THE FILE!\n\n");
      printf("PRESS RETURN TO CONTINUE. . .");
      getchar();
      }
    fclose(album_file);
    }
  else
    {
    printf("\n\nTHERE IS NO ALBUM FILE ON THE DISK!\n\n\n");
    printf("PRESS RETURN TO CONTINUE. . .");
    getchar();
    }
}

title_search()
{
int found_one;
char title[25];

v_clrwk(work_stn_hndle);
v_curhome(work_stn_hndle);
printf("What titles do you wish to search? ");
gets(title);
if ((album_file = fopen(ALB_FILE,"r")) != NULL)
  {
  found_one = FALSE;
  while (fread(&an_album,sizeof(struct album_type),1,album_file)==1)
    if (strcmp(an_album.title,title) == 0)
    {
    found_one = TRUE;
    disp_the_record();
    }
  if (!found)
    {
    printf("\n\nTHAT TITLE IS NOT IN THE FILE!\n\n");
    printf("PRESS RETURN TO CONTINUE. . .");
    getchar();
    }
  fclose(album_file);
  }
else
```

```
    {
    printf("\n\nTHERE IS NO ALBUM FILE ON THE DISK!\n\n\n");
    printf("PRESS RETURN TO CONTINUE. . .");
    getchar( );
    }
}
year_search( )
{
int found_one;
char year[5];

v_clrwk(work_stn_hndle);
v_curhome(work_stn_hndle);
printf("What year do you wish to search? ");
gets(year);
if ((album_file = fopen(ALB_FILE,"r")) != NULL)
    {
    found_one = FALSE;
    while (fread(&an_album,sizeof(struct album_type),1,album_file)==1)
      if (strcmp(an_album.year,year) == 0)
        {
        found_one = TRUE;
        disp_the_record( );
        }
    if (!found_one)
      {
      printf("\n\nTHAT YEAR IS NOT IN THE FILE!\n\n");
      printf("PRESS RETURN TO CONTINUE. . .");
      getchar( );
      }
    fclose(album_file);
    }
  else
    {
    printf("\n\nTHERE IS NO ALBUM FILE ON THE DISK!\n\n\n");
    printf("PRESS RETURN TO CONTINUE. . .");
    getchar( );
    }
}
```

The album data base program is very modularized and lends itself to the addition of new options and modules. Each of the search routines is almost identical, except, of course, for the field being searched. When you wish to stop adding albums to the file, just enter a dollar sign ($) and control will

return to the main menu. The extra field in the structure may be used to specify the condition of the album, the owner of the album, etc. Be sure to log some or all of your albums so that you can see how easy it is, then to see if you have any albums by a particular artist, from a certain year, etc.

Summing Up

The date minder program can definitely be a help to you and your family if you are prone to forgetting important dates. The program is also set up in a modular manner so that you could easily add more options to the main menu—like viewing the next 60 days' events or maybe even the entire file. Because of the modular structure of the program (i.e., each task has its own separate routine), all you would need to do is add more options to the menu in **display_menu**( ) (and make sure they are accepted as valid in the "if" and **switch** statements thereafter) and add the appropriate functions to the program. The baseball and album programs are both quite modular and can therefore easily be customized to your own unique needs.

We sincerely hope that the information you have learned in this and all the preceding chapters are all you need to write your own applications. Even though you are now finished with this book, you should always keep it handy when working in C on your Atari ST as a quick reference guide.

Glossary for Review

modular structure—the structure used in all the programs in this chapter, where each separate task has its own routine; more tasks can be added by simply developing additional routines

multidimensional arrays—arrays that have two or more indices (such as the months variable in the date minder program)

Quiz

1. Why couldn't we simply have added 1 to the current month to get the value of the next month in the date minder program?
2. Why do we always check the value of the **at_bats** field before calculating the batting average in the baseball averages program?
3. How are invalid entries handled in the main menu of the album data base program?

Appendix A

Reserved Words

auto	break	case
char	continue	default
do	double	else
entry	extern	float
for	goto	if
int	long	register
return	short	sizeof
static	struct	switch
typedef	union	unsigned
while		

Appendix B

Answers to Quiz Questions

Chapter 1

1. This compiler follows the rules of C as laid out in Kernighan and Ritchie's *The C Programming Language* (1978, Prentice-Hall).
2. Power and portability.
3. This statement is a comment and is therefore ignored by the compiler, even though the **printf** command is misspelled.

Chapter 2

1. Generally, any identifier created with a **#define** statement should be specified in lowercase letters.
2. **#include ⟨filename.ext⟩** and **#include "filename.ext".**
3. Both data types occupy 4 bytes of memory.
4. **'\0'** is the string terminator.

Chapter 3

1. The ampersand (**&**) is missing before the **num** parameter.
2. **getchar( )** may be used to get input from the user, and **putchar( )** may be used to display the input from **getchar( ).**

3. If the **num** variable is not zero, the first message is printed; otherwise, the second is printed.

Chapter 4

1. The "do-while" loop will always execute at least one iteration, whereas the "while" loop may not execute any if the condition is not true.
2. **num++** increments **num** after its current value (post-incrementing) has been used in this statement, and **++num** increments **num** (pre-incrementing), and this new value is used in the statement.
3. The loop will not execute any iterations, since **i** is initialized to 5, and the condition for looping states that it must be less than 5 to perform another iteration.

Chapter 5

1. Global variables may be accessed and changed by any function, whereas local variables are limited to change by only the function in which they are declared.
2. This declares **number** to be a pointer to an int value.
3. Passing by address permits the called function to modify the actual parameters that the calling function sent. Passing by value, on the other hand, does not allow this, since a copy, and not the actual value, is passed to the routine.

Chapter 6

1. A. Atari B. 5
   C. 12 D. nonzero
   E. zero F. zero
2. Any test sent to the screen via **printf** will begin where the cursor was positioned before the call to **v_clrwk**.
3. The parameters in **wind_create** represent the maximum size of the window, whereas they represent the actual initial size in **wind_open**.

Chapter 7

1. Because they vary throughout the program depending on the number entered.

2. They are broken apart so that they may be called independently of each other (e.g., what if you only wanted to do one conversion?).
3. Because you should always have at least one entry.
4. Yes. All the numbers from zero through nine.

Chapter 8

1. Simply put, a file is a collection of records, and a record is a grouping of fields!
2. Because the purpose of a record/structure is not to try and centralize all your data structures, but rather to group together all related items (like the name, position, etc., in our baseball program).
3. It was used to remove any possible "garbage" that may have been left over in memory from any other variables that may have been using that memory location. It is not necessary for you to understand why this "garbage" might exist; however, it is important for you always to initialize your arrays just as you would initialize any other variable.

Chapter 9

1. The entire collection would be called the "file," each album would be called a "record" or "entry," and each song would be a "field" for that particular "record."
2. We used the append ("**a**") mode for writing to the file in the **fopen** function.
3. You could use a constant name as declared with **#define**; this would limit your capability to have more than one file, however, since you would have to change the **#define** statement and recompile and relink every time you wanted to use a different file name.

Chapter 10

1. It forces the programmer to think through the program logic more thoroughly without having a debugger to fall back on for help.
2. **C IS FUN**

Chapter 11

1. The SHADOW effect would still be in effect, and, therefore, the text displayed in the **v_gtext** after **do_text()** would be displayed with the shadowed effect.

2. If the mouse is clicked at the origin on the second point, a line is drawn from the first point to the origin. Remember that the only time a click at the origin causes the program to stop is when it is the first of the two clicks.

Chapter 12

1. That would be fine and is indeed what we do, except for when the current month is December, and the next month is therefore January; adding 1 to 12 would be 13, not 1!

2. If we didn't check the value of the **at_bats** field, it would be possible to try to divide by zero, which would cause the program to abort.

3. The default option in the **switch** statement performs nothing, so when the "while" loop starts again, the screen is simply redisplayed, and the menu once again prompts for a selection.

Index

Accumulator loop, 55
Advanced data structures/concepts
  arrays, 121–30
    parallel, 124–25
    simple, 121–24
  numeric types, 120–21
  recursion, 130–33
  structures/files, 125–30
appl_exit( ), 92
Application Environment System (AES), 5
appl_init( ), 91
Arrays
  application use, 127–30
  multidimensional arrays, 195
parallel arrays, 124–25
  simple arrays, 121–24
  versus files, 137
Atari ST C compilers, 3–5
  Graphics Environment Manager (GEM), 4–5
    Application Environment System (AES), 5
    Virtual Device Interface (VDI), 5

Baseball averages program, 182–87
Baseball lineup program, 127–30
BASIC language
  versus C language, 2–3
  interpreter, 3
  origin of, 2

Binary search, 104
Bits, definition of, 19–20
Byte, definition of, 19–20

Central library, 86–87
CIPHTEXT file, 160
C language
  versus BASIC 2–3
  data types, 19–24
  versus Logo, 2–3
  origin of, 2
  programs
    elements of, 16–18
    expressions, 27–28
    mathematical order of operation, 28–29
    program formatting, 31–32
    simple C arithmetic, 29–31
    storage classes, 25–27
  syntax, 16
  variables, 18–19
    assignment of, 24–25
Compilation process, Megamax C compiler, 7–8
Conditional statements, 39–49
  "if" statement, 40–42
    common errors with, 42–44
    nesting of, 44–46
  logical operators, 46Logical operators, 46

?:(conditional) operator, 46–4946
switch alternative, 47–49
Cosine function, 87
C Programming Language (Kernighan/Ritchie), 2

Data types
characters, 22–23
floats, 21–22
identifiers, 24
int, 19–20
strings, 23–24
Date minder program, 176–82
Debugging
definition of, 152
echo printing, 153
tips, 152–53
Display phone book program, 148–49
"Do-while" statement, 65–66
versus "while" statement, 66

Echo printing, 153
Encipher/decipher program, 158–60
Expressions, 27–28
External storage class, 26–27

Family budget program, 65–66
File analysis
encipher/decipher program, 158–60
hex/character file dump program, 153–56
Files, 125–30, 136
CIPHTEXT file, 160
handling library routines, 138–43
phone book program and, 139–43
versus arrays of structures, 137
Formatting, 31–32
"For" statement, 59–65
defintion of, 59
flexibility of, 63
Functions
ASCII-to-integer function, 87–88
components of, 73
cosine function, 87
definition of, 73
functions program, 74–75
getc( ) function, 156
gets function, 88
macros, 89–90
mathematical functions, 87–88
placement of, 73–74
return statement, 80–82
sine function, 87
square root function, 87
strcat function, 88–89
strcmp function, 89
string functions, 88–89
strlen function, 89
uses of, 74–76
using variables with, 76–79
global versus local variables, 76–78
value parameters, 78–79
variable parameters, 82–84

Gas mileage calculation program, 29–30
getc( ) function, 156
getchar function, 39
gets function, 88
global variables, 76–77
goto statement, 67–68
graf_handle( ), 91
graf_mouse( ), 92–93
Graphics
Atari ST graphics, 164–71
graphics example program, 164–67
v_bar routine, 170
v_circle library routine, 169
v_gtext function, 168–69
v_pline library routine, 169
v_recf function, 170
vsf_interior function, 168
vsl_color routine, 167, 168
vst_effects function, 170

mouse drawing program, 176–82
vq_mouse function, 173–74
Graphics Environment Manager (GEM), 4–5
Application Environment System (AES), 5
functions of, 90–93
appl_exit( ), 92
appl_init( ), 91
graf_handle( ), 91
graf_mouse( ), 92–93
v_clrwk( ), 93
v_clswk( ), 92
v_curhome( ), 93
v_opnvwk( ), 92
vq_mouse, 173–74
windows, 93–97
Virtual Device Interface (VDI), 5

Hexadecimal conversion program, 110–14
ASCII conversion values, 156–58
program explanation, 116–17
Hex/character file dump program, 153–56

"If" statement, 40–42
common errors with, 42–44
"if" statement program, 41–43
nesting of, 44–46
Interpreters, 2–3

Lattice C compiler, 3–4
Library features
central library, 86–87
Graphics Environmental Manager (GEM) functions, 90–93
windows, 93–97
standard C functions, 87–90
macros, 89–90
mathematical, 87–88
string, 88–89
Linking, Megamax C compiler, 10–11
Local variables, 76–77
Logical operators, 46
Logo, versus C language, 2–3
Loop counter variable, construction of, 55
Looping structures
accumulator loops, 55
counting loops, 54–55
"do-while" statement, 65–66
"for" statement, 59–65
definition of, 59
flexibility of, 63
goto statement, 67–68
infinite loops, 53
loops that sum, 55–59
nested loops, 66–67
summing loops, 55
"while" statement, 52–54

Macros, 89–90
Mark Williams C package, 3, 4
Mathematical functions
ASCII-to-integer function, 87–88
cosine function, 87
sine function, 87
square root function, 87
Mathematical order of operation, 28–29
Megamax C compiler
compilation process, 7–8
compiler package, 3, 4
float data types, 22
int data types, 21
use of, 5–12
linking, 10–11
microfloppy disks, 5
program entry, 5–6
saving programs, 6–7
Merging, 143–50
definition of, 150

display phone book program, 148–49
sort/merge phone book program, 144–47
Metric conversions program, 100–4
program explanation, 103–4
Microfloppy disks, Megamax C compiler, 5
Modular programming, 72
Modular structure, 195
Mouse
mouse drawing function, 176–82
vq mouse function, 173–74
Mouse drawing program, 176–82
Multidimensional arrays, 195

Nested loops, 66–67
conditional statements, 44–46
nested loop program, 67
NULL, definition of, 139, 150
Number guessing program, 104–10
program explanation, 108–10
Numeric types, advanced, 120–21

Parallel arrays, 124–25
Phone book program, 139–43
Pixels, 95
Polling, 173, 174
Post-/pre-decrementing, 57–59
Post-/pre-incrementing, 57–59
Printf function, 34–37
Program entry, Megamax C compiler, 5–6
Program formatting, 31–32
Programs
accounting program, 60–61
with less code, 62, 63–64
baseball averages program, 182–87
date minder program, 176–82
difference between ++var/var++ program, 58
elements of, 16–18
family budget program, 65–66
first "while" loop program, 53
functions program, 74–75
gas mileage calculation program, 29–30
"if" statement program, 41–43
metric conversion program, 100–4
program explanation, 103–4
nested loop program, 67
number guessing program, 104–9
program explanation, 108–9
pay including overtime program, 45–46
phone book program, 139–43
products in a loop calculation, 54
record album data base program, 187–95
recursion program, 131–32
student's grade point average program, 30–31
summing program, 55–57
window display program, 94–95
Putchar function, 37

Record album data base program, 187–95
Record data structure, 125–30
Recursion, 130–33
Recursion Program, 131–32
Register storage class, 26
Reserved words, 197
Return statement, 80–82

Saving programs, Megamax C compiler, 6–7
scanf function, 37–38
Scope rules, 76
Sign bit, 120
Simple arrays, 121–24
Simple C arithmetic, 29–31
Sine function, 87
Sorting
definition of, 150

display phone book program, 148–49
sort merge phone book program, 144–47
Square root function, 87
Statements
conditional statements, 39–49
"do-while" statement, 65–66
"for" statement, 59–64
getchar function, 39
goto statement, 67–68
"if" statement, 40–42
printf function, 34–37
putchar function, 37
scanf function, 37–38
"while" statement, 52–54
See also Conditional statements.
Static storage class, 26
Stepwise refinement, 72
Storage classes, 25–27
for automatic variables, 25–26
external storage class, 26–27
register storage class, 26
static storage class, 26
Strcat function, 88–89
Strcmp function, 89
String functions
gets function, 88
strcat function, 88–89
strcmp function, 89
strlen function, 89
strlen function, 89
Structured design, 72
Structures, 125–30
application use, 127–30
Student's grade point average program, 30–31
Summing loop, 55
Summing program, 55–57
Switch alternative, 47–49
Symbolic constants, 17
Syntax, 16

Top-down design, 72

Unsigned modifier, 20–21

Variable parameters, 82–84
Variables, 18–19
assignment of, 24–25
global variables, 76–77
identifiers, 24
local variables, 76–77
used with functions, 76–79
value parameters, 78–79
variable parameters, 82–84
v_bar routine, 170
v_circle library routine, 169
v_clrwk( ), 93
v_clsvwk( ), 92
v_curhome( ), 93
v_gtext function, 168–69
Virtual Device Interface (VDI), 5
v_opnvwk( ), 92
v_pline library routine, 169
vq_mouse, 173–74
v_recf function, 170
vsf_interior function, 168
vsl_color routine, 167, 168
vst_effects function, 170

"While" statement, 52–54
versus "do-while" statement, 66
"while" loop program, 53
Windows, 93–97

>$19.95

You Already Own a Powerful Personal Computer . . . Now Put Its Power to Work for You with ***Learning C on the Atari ST***

If you want to create your own programs in C on the Atari ST personal computer, ***Learning C on the Atari ST*** is for you. Beginning and experienced programmers alike will appreciate the easy-to-use, readable style of ***Learning C on the Atari ST.*** Because the book describes the C programming language so thoroughly, beginners will be able to write code immediately. Advanced programmers will appreciate the detailed coverage of the unique features of the ST.

This well-organized book includes:

- Recommended C compilers for the Atari ST
- What the Graphics Environment Manager is, and how you can use GEM to take advantage of the ST's powerful capabilities
- The proper syntax and structure of a C program, what variables are, how to comment and define constants
- How to write statements and routines as well as how to construct a loop
- Various C functions and how to write more sophisticated programs that use files, multi-dimensional arrays, and pointers
- Popular graphics routines, and how to design programs that use a mouse.

And, all programs are written so that they can be easily converted to most ST C compilers on the market.

Because each chapter begins with objectives and ends with a review and quiz, you won't miss important information essential to a clear understanding of the C language. With ***Learning C on the Atari ST,*** you'll build solid C programming skills. And because this guide is so easy to use, you'll want to keep it handy as a reference guide.

Joseph Boyle Wikert is a systems analyst for NCR Corporation. He is also coauthor of *Using Your Macintosh: Beginning Microsoft BASIC and Applications* and *Learning Macintosh Pascal* (published by Scott, Foresman). He currently lives in Miamisburg, Ohio.

Scott, Foresman and Company

ISBN 0-673-18738-1