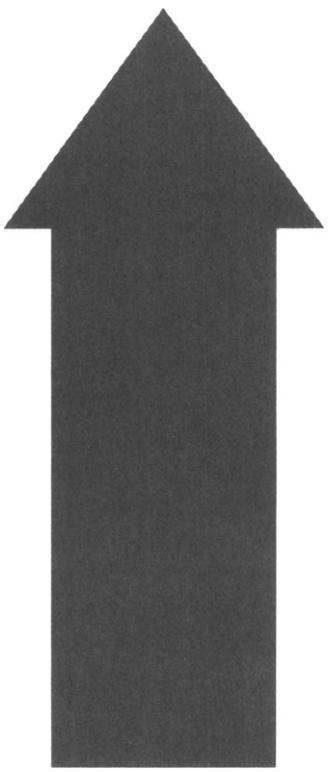
Mastering C Pointers
Second Edition

Tools for Programming Power



Robert J. Traister

Robert J. Traister & Associates
Front Royal, Virginia

AP PROFESSIONAL
A Division of Harcourt Brace & Company

Boston San Diego New York
London Sydney Tokyo Toronto

AP PROFESSIONAL
955 Massachusetts Avenue, Cambridge, MA 02139

An Imprint of ACADEMIC PRESS, INC.
A Division of HARCOURT BRACE & COMPANY

United Kingdom Edition published by
ACADEMIC PRESS LIMITED
24-28 Oval Road, London NW1 7DX

Library of Congress Cataloging-in-Publication Data

Traister, Robert J.
Mastering C pointers : tools for programming power / Robert Traister. — 2nd ed.
p. cm.
Includes index.
ISBN 0-12-697409-8
1. C (Computer program language) I. Title.
QA76.73.C15T697 1993
005.13'3—dc20 93-10540
CIP

Printed in the United States of America
93 94 95 96 EB 9 8 7 6 5 4 3 2 1

This book is lovingly dedicated to Lin and Nita Courtenay,
whose friendship is cherished

LIMITED WARRANTY AND DISCLAIMER OF LIABILITY

ACADEMIC PRESS, INC. ("AP") AND ANYONE ELSE WHO HAS BEEN INVOLVED IN THE CREATION OR PRODUCTION OF THE ACCOMPANYING CODE ("THE PRODUCT") CANNOT AND DO NOT WARRANT THE PERFORMANCE OR RESULTS THAT MAY BE OBTAINED BY USING THE PRODUCT. THE PRODUCT IS SOLD "AS IS" WITHOUT WARRANTY OF ANY KIND (EXCEPT AS HEREAFTER DESCRIBED), EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY WARRANTY OF PERFORMANCE OR ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. AP WARRANTS ONLY THAT THE MAGNETIC DISKETTE(S) ON WHICH THE CODE IS RECORDED IS FREE FROM DEFECTS IN MATERIAL AND FAULTY WORKMANSHIP UNDER THE NORMAL USE AND SERVICE FOR A PERIOD OF NINETY (90) DAYS FROM THE DATE THE PRODUCT IS DELIVERED. THE PURCHASER'S SOLE AND EXCLUSIVE REMEDY IN THE EVENT OF A DEFECT IS EXPRESSLY LIMITED TO EITHER REPLACEMENT OF THE DISKETTE(S) OR REFUND OF THE PURCHASE PRICE, AT AP'S SOLE DISCRETION.

IN NO EVENT, WHETHER AS A RESULT OF BREACH OF CONTRACT, WARRANTY OR TORT (INCLUDING NEGLIGENCE), WILL AP OR ANYONE WHO HAS BEEN INVOLVED IN THE CREATION OR PRODUCTION OF THE PRODUCT BE LIABLE TO PURCHASER FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PRODUCT OR ANY MODIFICATIONS THEREOF, OR DUE TO THE CONTENTS OF THE CODE, EVEN IF AP HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

THE RE-EXPORT OF UNITED STATES ORIGIN SOFTWARE IS SUBJECT TO THE UNITED STATES LAWS UNDER THE EXPORT ADMINISTRATION ACT OF 1969 AS AMENDED. ANY FURTHER SALE OF THE PRODUCT SHALL BE IN COMPLIANCE WITH THE UNITED STATES DEPARTMENT OF COMMERCE ADMINISTRATION REGULATIONS. COMPLIANCE WITH SUCH REGULATIONS IS YOUR RESPONSIBILITY AND NOT THE RESPONSIBILITY OF AP.

Preface

The first edition of *Mastering C Pointers* was written in 1989. The purpose of the book was to provide a tutorial/reference source that was unique. Prior to its being published, there were no books on the market that dealt exclusively with C language pointers. Most books about C language just mentioned pointers or dedicated a small chapter to them. That was about the extent of it.

This author felt that an entire book should be written on the subject of pointers to clear up the large amount of misunderstanding about their purpose and uses, to establish a firm educational base in the beginning programmer's mind, and to remove a general fear that new C programmers and even those with moderate experience had exhibited.

Prior to publication of the first edition, the manuscript was reviewed by a professional C programmer hired by the publisher. This individual expressed a firm opinion that the book should not be published because "it offers nothing new—nothing the C programmer cannot obtain from the documentation provided with C compilers by the software companies."

This review was not surprising. The reviewer was of an opinion that was shared by, perhaps, the majority of professional programmers who have little knowledge of or empathy

for the rigors a beginning programmer must overcome to achieve a professional-level knowledge base of a highly technical subject.

Fortunately, that reviewer's objections were disregarded, and *Mastering C Pointers* was released in 1990. It was an immediate success, as are most books that have absolutely no competition in the marketplace. This was and still is the only book dedicated solely to the subject of pointers and pointer operations using the C programming language.

Having been released in several foreign language editions as well as in the original English, *Mastering C Pointers* has proved this author's original contention that the growing number of C programmers was hungry for a thorough but simple explanation of pointers.

However, computer languages are not static. They change and change often, usually for the better. Some four years after the original manuscript was completed, it is time for a second edition. This greatly revised and enhanced text fully embraces the ANSI standard for the C programming language.

This edition still adheres to the friendly, chatty format of the original work. However, discussions and program examples have been redesigned to reflect the changes that have been brought about with the full adoption of ANSI C. Additional chapters have been added to microscopically explore C language structs and unions, pointers to pointers, and many other interesting topics. The basis behind the second edition was to retain everything that was still relevant from the first edition, to modernize all discussions and program examples, and to add many materials that are, today, necessary reading for any ANSI C programmer.

If you desire to learn more about pointers—why they exist, how they are used, and how they can be incorporated into new and existing programs to enhance program power—in an environment of ANSI C, then *Mastering C Pointers*, 2nd edition, should provide you with pleasant comprehensive reading that will lead to proficiency in C pointers.

Robert J. Traister

Chapter 1

C Language and Pointers—A Personal Background

The C programming language was developed in the early 1970s by a small team of employees at Bell Laboratories. Shortly before then, Bell had been involved in a software project with another major corporation, collaborating on a multi-user environment called MULTICS. However, the collaboration was ended suddenly with the MULTICS project still unfinished. One of the Bell employees involved with this project, Dennis Ritchie, was frustrated with the inefficiency of existing high-level languages and by the time required to write programs in assembly language. After termination of the MULTICS collaboration, Ritchie and a handful of other Bell employees began an informal development project as an extracurricular activity. The end result was the UNIX operating system.

As popular as UNIX is today, an even more important product was the C programming language. This was the personal language of the development team, a language developed specifically to answer the challenges of writing the UNIX operating system. UNIX was written almost entirely in C and contains a built-in C compiler. Due to the C/UNIX link, future C programming was accomplished almost exclusively within the environment of UNIX.

C holds a very special place in the classes of languages. Assembly language is a low-level type that offers the highest versatility but also incorporates tedious coding as the price for this feature. BASIC is a high-level language. It offers far less versatility but features faster program construction via the high-level statement and function operations inherent in it and in other high-level languages.

The C programming language is rated as a moderate level language. It offers much of the versatility of low-level languages and many of the fast-coding abilities of the high-level languages.

C language, while unique, does have its roots in other languages. Its earliest influences can be traced to ALGOL, and it can be tracked through the following sequence of languages.

1960	ALGOL 60
1963	CPL
1967	BCPL
1970	B
1972	C

C was created to save system-level programmers considerable time and effort, especially in writing powerful code constructs. Many programmers originally viewed C as a shorthand approach to assembly language programming.

Until 1982, the C programming language was a highly specialized tool used by a small number of professional programmers, but the C boom was about to take place! In the early 1980s, C was discovered by the major software houses, the originators of software packages for a new but fast-growing group of computer users that evolved along with the introduction of the personal computer. The new IBM PC was being snatched up by small businesses along with a growing group of hobby programmers.

C language began to make major inroads during the 1980s. In addition to its ease of use when compared with assembly language and its versatility, it was renowned as a very portable language. This description is most appreciated when C is compared with assembly language. An assembler for one type of microprocessor architecture is very different from the assembler for another type. Prior to the use of C language, most commercial programs were written in assembler. However, when software manufacturers had developed a winning program for one type of computer and wanted to offer the same program for another, most of the assembler code had to be completely rewritten. This meant many more programmer hours, higher end product costs, and loss of profits.

However, the C programming language is easily ported to another type of microprocessor with little or no source code changes. When this trait was discovered, C quickly became the main language tool of the professional software developer.

C is not a low-level language like assembler. One disadvantage of this trait lies in the fact that programs written in assembly language will usually execute faster than equivalent programs written in C. For this reason, program segments that are highly speed-critical will often be coded in assembly language and called from within the C program that makes up the major portion of the application. A high degree of portability is still retained by this mixing of languages, since the only portions that must be rewritten when bringing an established program up on a new type of computer are those small segments written in assembler.

The major code portion, when written in C, may be ported to the new computer in relatively unchanged form.

C Language Standards

The original standard for C language was contained in a book written by Brian Kernighan and Dennis Ritchie, *The C Programming Language.* Introduced in 1978, this was the sole C standard for many years and was commonly referred to as K&R C or UNIX C. However, the sudden popularity of the C programming language in the early and middle 1980s prompted the American National Standards Institute (ANSI) to develop its own standard for C. The great majority of C compilers available today conforms to the ANSI standard, and the language is called ANSI C.

C language has changed considerably since its initial development. As is usually the case with language development and progression, ANSI C is now a larger language than it once was in terms of the standard function set. The language has lost some of its simplicity in exchange for a few added conveniences. However, ANSI C is still a powerful, portable, and relatively unencumbered language that saves programmers time and effort in exchange for very little sacrifice in speed and power when compared with assemblers and other languages.

The C programming language, since its introduction in the very early 1970s, progressed from the personal language of an elite group of programmers at Bell Laboratories to a somewhat exotic language of daring professional programmers and finally to the language of an ever-growing diversified number of average programmers. This latter group is comprised of you and me: professional programmers, novices, and those who work for the major software houses. C, then, has become an almost universal language. It has been described by many as the best language for microcomputers and by others as the best all-around language ever. These opinions are certainly arguable, but one fact is certain: C is an extremely popular object-oriented programming language that has made inroads into every fiber of the computer programming world.

The current popularity of ANSI C and the wealth of books, compilers, interpreters, and other forms of documentation are a far cry from the situation in the early 1980s. At that time, C was only a decade old, Kernighan and Ritchie were about the only ones who had dared write about C, and compilers were almost as scarce as documentation. In 1981, C was a mysterious language, its rites practiced only by a small group I have often referred to as a *priesthood.* And to those of us who were intent on learning this new language, this priesthood would often hand down the edict of "C is not for mortals!!!"

Indeed, my first encounters with the C programming language could only be described as tragic. When I was finally able to latch onto a C compiler that cost less than $600, I was astounded by the fact that it simply didn't work. No, I didn't receive a piece of defective media as would be the case with a damaged disk or improper copy, I got a compiler that simply didn't work. After weeks of calling the software company that supplied it (which

shall remain nameless), I got another version. Guess what? This one didn't work either. Finally on the third try, the compiler worked after a fashion. It still had an incredible amount of bugs, supported only a subset of the original C programming language, and was a complete abomination.

With this Frankensteinian C compiler, I was finally able to compile and run my first C program, the famous K&R "hello, world". While this was an exciting event for me personally, the fact that three months had elapsed since I had received the first compiler seemed to dictate that something else was needed. Even the third version of this compiler made it almost impossible to put to practice the little I had learned from Kernighan and Ritchie's *The C Programming Language.* Most of their teachings were simply not supported.

It was at this point that I vowed to give up on C completely. Indeed, I didn't try C again for almost eight months. However, a gratis copy of a C compiler from another software house arrived in my office one day. When I finally got around to using it (months later), I found that the scant few things I knew about C language were perfectly workable under this excellent compiler, and I progressed at a somewhat faster pace. Still, there were many missing features that I had become accustomed to as a BASIC programmer. Why were there no functions to clear the screen? Hey! They could at least have supplied a square root function. I would even have settled for a peek/poke routine that would allow me to address all of my PC's memory.

Well, those things were yet to come, and they eventually did. Fortunately for me, the expanded function sets did not arrive on the software scene before I had figured out how to write many of the most needed math and screen utility functions on my own, a learning experience that was invaluable. In those days, a C compiler that supported floating-point math operations as opposed to integers-only was considered a luxury item. Today, a C compiler that doesn't include the latest ANSI standard set of functions with its elaborate assortment of baubles, whistles, and bells is something to (rightfully) scorn.

The documentation field has grown with the C programming language over the last decade. In 1980, K&R's *The C Programming Language* was the only published programming source. This excellent work, by the authors' own admission, was not an introductory programming manual. It seemed so technical and foreign to me at the time that it's a wonder that I ever persisted. It was only after I had struggled for a year or more making every mistake possible in C that I had learned enough on my own to be able to fully appreciate the terse, dry wording of this book and to learn from it. Even today, I keep it nearby as a familiar reference source.

Today, the market abounds with ANSI C books for all skill levels and pursuits. If you're a BASIC programmer, then there is specialized help for you in one of many books. If you know ANSI C but want to know more, the intermediate-level books are also abundant. There are as many more aimed at the higher-level users. Yes, things have sure changed since the early 1980s as far as learning the C programming language is concerned.

Except in one area! The subject of C language pointers and pointer operations is still a

partially unexplored curriculum. Sure, most C programming books do offer a chapter on pointers, but these are usually very incomplete. This is quite understandable, because C pointers require at least an entire textbook for a worthwhile explanation. Several large volumes could easily be filled with explanations of the meaning, purpose, operation, and versatility of the pointer.

C pointer operations comprise one of the greatest stumbling blocks for persons learning the language. Most books that discuss C pointers only skim the surface. It would seem that those authors who know a lot about pointers often assume their readers do also. Those who know a lot about the other aspects of the language halfheartedly broach the subject just to round out the text contents. Both ways, not enough raw materials are presented. The result is a growing number of persons who call themselves ANSI C programmers but who really don't understand the full extent of the power pointers offer. Of course, *power programming* is an overused phrase. A good, basic knowledge of pointers allows the ANSI C programmer a relaxed and almost artistic method of handling tasks. You can certainly program in ANSI C without resorting to declared pointers, but you can't program as expressively or as efficiently as you can with pointers.

As is always the case, progress in C language compilers has lessened the chore of the programmer. With the advent of ANSI C, prototyping has finally come about, and this provides a relatively high degree of protection in passing arguments to functions. Without prototyping, data type requirements are more exacting. For instance, passing an int variable or constant to a function that requires a double type would result in garbage returns using older C compilers. With prototyping, the math functions are more versatile, since arguments of the incorrect data type are coerced (cast) to correct types where possible.

All of this is excellent from a user standpoint but probably not so useful to the student trying to grasp the concept of this very efficient, very expressive language. This outlook may be starting to sound like the proverbial "good ol' days" approach, wherein someone espouses the virtues of the one room-school, the outhouse, and the horse and buggy, but this is not my intention at all! Prototyping is a powerful new tool to ANSI C, but it should be used to enhance program efficiency, not to make up for bad programming methods. Bad programming technique will eventually catch up with you no matter how many safeguards are built into the language.

The only aim of this book is to reveal the C language pointer. It's not really that much different from any other type of variable. For the most part, all program examples will be short, sweet, and plentiful. We will be sticking solely to the very basics of the language. Once these are grasped, the fancy whistles and tin horns built into today's compilers will become even more valuable and will enhance each program you write.

You are encouraged to simply play with some of the ideas that are presented here, for it is through such play that programmers can explore new concepts and techniques. More realistically, recreational programming will yield a tremendous amount of mistakes that can be worked out at the reader's leisure and probably never forgotten. The impossible-to-cor-

rect error that is finally overcome is worth more from a tutorial standpoint than all of the documentation you are ever likely to read about the same program operation.

The model computer for researching this book was a Royal Oak Computers 80386 DX machine running at 40 mHz and equipped with 4 megabytes of RAM, a 170-megabyte hard disk, an XVGA color monitor, and MS-DOS 5.0. This is representative of a mid-range system in today's market. However, any of the program examples may be compiled and tested on just about any system from the original IBM PC to the latest 80486 and P-5 systems.

The model compiler used for testing all programs in this text was Borland C++ from Borland International. As is the case with many C++ compilers, Borland C++ can also operate in a pure ANSI C mode, the one used for documenting this book. This compiler (when used in ANSI C mode) offers all of the standard conventions of ANSI C and also allows its owner to move into the realm of C++.

The discussions in this book assume the reader has more than a conversational knowledge of ANSI C programming. This book is not a tutorial on the entire language. It is aimed at the advanced beginner to intermediate-level programmer who feels relatively comfortable writing C programs but who has a distance to travel when it comes to the topic of pointers.

It is quite possible, even likely, that many readers will have developed an erroneous concept of pointers and pointer operations. I speak from experience in this area. Due to the gloss-over approach taken by many forms of ANSI C documentation, such false impressions are easily acquired.

This book is written from a personal standpoint that may be best described as "I wish someone had told me this years ago when I was pulling my hair out trying to learn this aspect of the language." If I have any notable talent, it lies in never forgetting just how ignorant I once was. As I write these words, I think back to the years 1981 and 1982. I not only remember but still feel the frustration of trying to learn C by the "let's try everything until something works" method.

Remember, there were no primers for C in those days. There was no one to answer the most rudimentary questions. I finally came across a C programmer through a newspaper ad in a major city who agreed to write some simple programs involving C language file routines that I could use on a self-tutorial basis. What sounded like a reasonable rate of $16 per hour turned out to be $60 per hour. ("Gee Bob, you must have misunderstood me, I said $60 per hour.") In any event, I received about an hour's worth of the most rudimentary C programs, a bill for ten hours, and paid $600 for programs I can now write in twenty minutes. Sure, I was ripped off, but I learned from those programs. Maybe they were worth $6000! Today, you can learn one hundred times as much for less than $30 at your local book store.

This tale of woe and misery describes my roots in the C programming language. However, it has been invaluable to me from the standpoint of remembering my complete ignorance at that time and, most important, what I learned to overcome that state of affairs. I hope that these experiences will be of benefit to you as I attempt to impart some of the things I learned in these pages.

Chapter 2

A First Step to Learning Pointers

In ANSI C, *a pointer is a special variable that holds a memory address* and that's all it is! Pointers may be treated as standard variables in many ways, but *special* also means that they may behave a bit differently in other operations. In some ways they can be used in high-powered constructs like no other standard variable. Therefore, there are several familiar aspects that can be seen in pointers and several that will be unfamiliar to those programmers who are not accustomed to using them. Careless (though technically correct) use of pointers can easily result in a hodgepodge program that is nearly impossible to understand due to the misdirection.

It is highly advantageous to review some C programming basics, especially in regard to addressing and memory allocation, before moving headlong into the topic of pointers. Pointer operations cannot be fully understood until the general storage methods used in ANSI C are clearly fixed in the programmer's mind. These storage methods and rules are handled by the compiler and are relatively invisible to the programmer. By observing only what seems to be taking place, it is quite easy to form misconceptions, and these untruths and half truths multiply into total learning chaos through the faulty use of pointers.

Let's begin the discussion by using the following simple C program as an example.

```
#include <stdio.h>
void main(void)
{

     int i;

     i = 25;

     printf("%d\n", i);

}
```

In this code, *i* is declared a standard (auto) variable of type int. It is assigned a value of 25, and its contents are displayed on the screen using the printf() function. Nothing could be simpler. However, to fully comprehend pointer operations and to gain a far better understanding of the invisible processes that take place within a computer program, it is necessary to dissect each of the operations carried out under main().

What happens when the variable is declared? The answer is that space is set aside at a location in memory to allow storage of a 2-byte value necessary to store numbers within the standard integer range. (Note: 2-byte integer storage allocation is common for most MS-DOS machines running the most popular C compilers. The use of such compilers and systems is assumed throughout this book.) These 2 bytes are allocated somewhere in memory when the program is executed. The exact whereabouts of this memory location are unimportant to the operation of this particular program.

The assignment line tells the computer to store an object value of 25 in the 2 bytes allocated exclusively for variable *i*. This is stored as a 2-byte integer. The first byte will be equal to 25 while the second byte will be 0. This is the 2-byte integer coding for a value of 25. Other combinations allow the 2 bytes to represent a positive value (unsigned) of up to 65535 or positive/negative values of plus 32767 or minus 32768.

When the printf() function is executed, the object value in *i* (25) is displayed on the screen as an integer. This is quite simple, but it is desirable to go through each step of the program's operation, especially in regard to how memory is set aside for storage. This will become more important as the field of pointers is entered.

To repeat an earlier step in the program, the declaration of *i* as a variable of type int causes 2 bytes of private storage space to be set aside exclusively for this variable. We can find out exactly where this storage space is located in memory by using the unary ampersand operator (&), more appropriately called the *address-of* operator. The program that follows demonstrates the use of this operator.

```
#include <stdio.h>
void main(void)
{

     int i;

     i = 25;

     printf("%u\n", &i);

}
```

In the printf() function argument, preceding the variable name (*i*) with the ampersand causes the memory location, also called the *starting address* of the variable, to be returned instead of the value of the integer coded into this memory location. This gets into what are generally termed *lvalue* (left value) and *rvalue* (right value) designations. One is the memory address of the variable. The other is the object value stored at that address. The *&i* designation is a *pointer* to the address of storage set aside for the variable. *&i* will always point to the address of storage that was automatically allocated to hold the contents of variable *i*. This address cannot be changed.

Note that the %u conversion specification is used in the format string argument to printf(), which causes the memory address to be displayed as an unsigned integer. This is necessary because the address is probably at the high end of the 64K segment used by most small-memory model C compilers and is, thus, beyond the range of a signed (standard) int. The actual address returned by *&i* can vary and depends upon machine configurations, operating systems, and memory usage. Regardless of the value returned, this is the memory location that has been reserved for the exclusive storage of values assigned to variable *i* in the sample program under discussion.

With the test hardware/software used for researching this book, the sample program will return a value of 65514. This means that bytes 65514 and 65515 within the 64K segment of memory were set aside exclusively for storing the integer values assigned to variable *i*. Referring to the first program example where *i* was assigned a value of 25, peeking into memory locations 65514 and 65515 would yield values of 25 and 0, respectively. Remember that an int variable is allocated 2 bytes of storage using most small-memory model compilers designed for MS-DOS systems.

This 2-byte coding (25 and 0) yields a true value of 25 decimal. If the value assigned to variable *i* were changed to 1990, then the 2-byte coding would be 198 and 7, respectively, that is, the first byte assigned to variable *i* is equal to 198 while the second byte is equal to 7.

The point to remember at this stage in the discussion is that *i* is a declared variable of type

int and that *&i* is really a constant (at least it is during any single run of the program under discussion). It cannot be changed as it is directly tied with variable *i* and equates to a value that names the start of the address location set aside for the exclusive storage of assignments to this variable. In other words, we cannot reassign *&i* to some other value. By definition and design, it is always equal to the start of memory assigned to the variable. Obviously, the following expression is absolutely illegal.

```
&i = 43681;
```

&i can do only one thing. It returns the constant that names the address of storage for variable *i*. This is a very important fact whose weight will be realized as we move on.

For a correct and useful understanding of pointer operations, it is essential to know that there are two distinct values associated with all variables. (Note: Register variables are an exception and do not hold closely to the direction of this discussion.) The one with which we are most familiar is the object value or the value that is assigned by the programmer as the following statement demonstrates.

```
i = 25;
```

In this example, the object value is 25, and it is stored at the memory locations specifically set aside for variable *i*. The second value associated with all variables is the memory address, the numerical location in memory where storage is allocated for the variable. The designation, *&i*, should not be thought of as an extension of the variable. It should be considered a constant. Unlike a variable, this cannot be reassigned. It is fixed. The address-of operator (&), when used with the variable name, causes its address to be returned.

Referring to an earlier definition of a pointer (a special variable that contains a memory address and nothing else), *&i* would seem to qualify. Again, its value (in regard to memory location) is fixed. The *&i* designation can be used as the argument to a function that is looking for a pointer argument, and it does point to a place in memory. But this is only one expression of a pointer in ANSI C. Such a designation is not the type of pointer that is often referred to in C operations. We could cloud things further by referring to it as a *fixed* pointer, as it is inextricably fixed to the memory location of variable *i*. This will be explained in more detail.

Newcomers to the C programming language, especially BASIC programmers, are often cautioned about the dangers of uninitialized (unassigned) variables. In the latter language, the object values in all numeric variables are initially equal to 0. When these variables are created, the memory location assigned to each is automatically nulled (set to 0). String values begin life with values of NULL (0) as well.

In ANSI C, the situation is completely different. When an auto variable is declared, only one basic operation takes place. Storage space is allocated exclusively for that variable. The

allocated bytes at this memory location are not cleared or reset to 0 as is the case with the BASIC language. Whatever bytes happened to be present at the address where storage is allocated will remain. Therefore, when our int *i* variable was declared in an earlier program, the contents in the bytes at memory location 65514 and 65515 (2 bytes per int) could have been any value from 0 to 255. If the programmer should mistakenly assume that the starting value in *i* is 0, then all sorts of problems can occur. The initial value of a declared variable, such as int *i*, has as much chance of being equal to 0 as it does of being equal to any other number within the legal integer range (-32768 to +32767).

This is especially important to remember when dealing with declared pointers, because they can point to any memory location when they are first created. All a pointer can do is store or return an address in memory. What you do with or at this memory address is up to you.

When a standard auto variable is declared, it must be remembered that the storage space that is automatically set aside for it can be considered safe storage. Different portions of computer memory are reserved for different things when a program is loaded and executed. Some of these areas deal with program management and interface with the operating system. This area is the framework that allows the program to be properly executed. Storage for variables is not allocated in such areas, because to do so would interrupt the framework and result in a program crash. Therefore, a certain area of memory is set aside exclusively for variable storage. When dealing with declared pointers, we will learn that these latter variables are not restricted to pointing to safe areas like other types of variables and can move anywhere in memory. This can result in execution catastrophe if the programmer is not on the ball.

The Little Man Approach

The ideal programming book, according to publishers who sign authors to write them, is "of a high enough technical level to appeal to the professional programmer but basic enough to be highly instructive and easy to understand by the novice trying to get a foothold on the subject." As one who is signed to write such books, I consider these requirements to be on a par with designing a full-fledged competition automobile that can out-accelerate any other race car and yet still gets 43 miles to the gallon for those persons who want an economy car. The two are not simultaneously possible. However, one must still make an attempt to offer technical materials to a broad range of readers.

As one who has learned several, diverse computer languages, I personally prefer to be treated like a complete novice and to read materials aimed at the beginner. I would rather a book be aimed at a readership just below my current technical level of expertise than to one that is slightly elevated beyond this level. I became very disgusted in my first attempts to learn LISP because I had to search my software manual and my technical books for several hours before I finally found the needed function to write to the screen. To my way of thinking, this should have been one of the first subjects discussed, because it allows the readers to do a bit of experimenting on their own and to see the results of their efforts.

Because of the way I like to be taught, the "Little Man and his Knapsack" will grace the pages of several chapters in this book. This approach is intended to get the reader quickly up to a higher level in understanding C pointers and not to talk down on the populace from some lofty tower.

Taking up our discussion to this point, the single variable, int *i*, is likened to a little man with a knapsack on his back. In this knapsack, he has two compartments, each capable of holding a byte of information. He lives in a neighborhood that contains hundreds of different houses. Some are vacant, while others are homes to more little men, sometimes there is only one to a household, sometimes more. These other neighbors also have knapsacks, some larger and some smaller than our first little man.

For instance, the man that represents a long integer has a larger knapsack that contains four separate compartments, each holding a byte of information. The largest knapsack is owned by Mr. Double and has eight compartments for holding a double-precision floating-point value. In representing character arrays, many little men, each with a knapsack containing only one compartment to hold 1 byte of information, are domiciled at the same residence.

The following program will begin to explain our neighborhood designations.

```
/* SAMPLE */
#include <stdio.h>
void main(void)
{

     int i;
     double d;

     long ii;

     i = 34;
     d = 3712.8876;
     ii = 123000;

     printf("%u  %u  %u\n", &i, &d, &ii);

}
```

The neighborhood for this program consists of three little men and three houses. *Mr. i*, *Mr. d*, and *Mr. ii* all live in the same neighborhood, each in a different house. Problem! We know that they live on SAMPLE street in this neighborhood, but we don't know what their respective addresses are. The printf() function, as used in this program, can be likened to a neighborhood directory. The *&i*, *&d*, and *&ii* designations are the addresses themselves.

Therefore, at the beginning of each street in our fictitious neighborhood, there is the PRINTF directory with a movable arrow, or POINTER, with which to point to the names of Mr. i, Mr. d, and Mr. ii. When the arrow is aimed at Mr. i, the directory displays his house address of 65504 Sample Street. When the pointer is aimed at the name of Mr. d, the address of 65506 is displayed on the directory. Going through the same mechanics for Mr. ii, we find that he lives at 65514 Sample Street. (Note: These addresses are relatively arbitrary. The point to remember is that whenever the pointer that is a part of the street directory is aimed at a name, the address of that "person" is returned.)

Expanding on the previous program, we arrive at the following.

```
/* SAMPLE */
#include <stdio.h>
void main(void)
{

     int i;
     double d;
     long ii;

     i = 34;
     d = 3712.8876;
     ii = 123000;

     printf("%u  %u  %u\n", &i, &d, &ii);
     printf("%d %d %d\n", sizeof(i), sizeof(d),
sizeof(ii));

}
```

The added statement using printf() provides another type of directory. The sizeof() operator allows us to list the number of compartments each little man has in his knapsack. This separate directory lets us know that Mr. i has 2 compartments in his knapsack, Mr. d has 8 compartments, and Mr. ii has 4 compartments. These, of course, correspond to 2, 8, and 4 bytes of storage space for integer, double-precision floating-point, and long integer values.

Summary

This chapter was a basic introduction to the essential processes that occur when a variable is declared and assigned. Upon variable declaration, an exclusive area of memory, allocated specifically for storage to variables, is set aside. The number of bytes retained was dependent

upon the type of variable and can be as large as 8 bytes when dealing with auto variables in a typical MS-DOS system.

Each variable is associated with two distinct values. The most common is the object value, which is assigned to this variable by the program. The second value is the memory address of the variable. The latter can be obtained by using the address-of operator (&) in conjunction with the variable name. Any assignments made to the variable are stored in the bytes that begin at this memory address. While it is certainly easy and necessary to change the object value, since this is what is generally considered to be the true variable value, it is not possible to change the memory address of a declared variable. This is fixed and can be thought of as a constant after the variable has been declared.

Auto variables in ANSI C, the type most often used in this programming language (as opposed to register and static variables), can be equal to any legal value when first declared and before any object values are assigned in the program. This is true because, in C, declaring a variable simply causes the compiler to set aside storage space. The space reserved for this variable is not cleared to zero as is the case in some other languages.

The space set aside for the exclusive storage of each declared variable lies in a safe area of memory. This is a product of the compiler's memory management system. This means that the area of memory used for this purpose is not shared by other services that could overwrite the variable's object values or be overwritten by object value assignments to the variable. Likewise, no two variables will be allocated storage at the same address (again, when dealing with the types of variables discussed to this point). These areas of memory, then, are exclusive and safe, at least from intrusion by other standard variables.

Important! A standard variable can be assigned any legal object value. When dealing with numeric variables as have been discussed in this chapter, this means object assignments within the normal numeric range for the type of variable declared. Variable *i*, for instance, can be assigned a value of 14 and then reassigned a new value of 234. This can go on ad infinitum. However, the memory address of that variable is fixed. It cannot be reassigned. If storage for this variable is allocated at memory location 65514, then this is that variable's fixed address for the duration of the program. It cannot be changed.

I realize that most of this discussion may be old hat to many readers. However, I have also worked with C programmers who have been at this game for quite a while and who still do not visualize the processes that take place during simple declaration and assignment operations. If you do not completely understand all of the discussions in this chapter, then please study these materials further, because their content is essential to the full understanding of the information in the rest of the book.

Chapter 3

A Second Step to Learning Pointers

Previous discussions have touched on the processes that take place, invisibly and internally, when declaring auto variables in C language. This forms the start of a learning base from which to proceed into the full subject of C pointers. This chapter expands upon that base, broadens it, and makes for a smooth transition into the subject of declaring pointers and using them to the best programming advantage.

C Language Strings and Char Arrays

When dealing with character strings in ANSI C, we come closer to pure pointer operations. However, C strings, or more appropriately, arrays of characters, also bear a close similarity to working with numbers and with numeric variables. Therefore, the creation and use of character arrays in ANSI C is the next logical step in exploring pointers.

The following program will aid in this discussion.

```
#include <stdio.h>
void main(void)
{
        char c;
        c = 65;
        printf("%c\n", c);
}
```

This program displays the letter *A* on the monitor screen, but as before, let's run through the invisible processes that take place when this simple program is executed.

The declaration line states that variable *c* is of type char. The majority of MS-DOS microcomputers running popular C compilers allocate 1 byte of storage for a char data type. A value of 65, as is the case here, represents the upper case A while B is 66, C is 67, etc. The assignment line uses the ASCII numeric value, but this line could just as easily have been written in the following manner.

```
c = 'A';
```

In C terminology, the 'A' designation means exactly the same thing as 65. Indeed, 'A' is automatically stored by the compiler as decimal 65.

The only reason the 'A' is displayed on the screen as a letter instead of a number is due to the %c designation in the printf() format control string. This specifies that the value in its argument, *c*, be displayed as an ASCII character.

As was discussed in Chapter 2, the address of the 1 byte of storage allocated to char *c* can be returned by adding the following program line.

```
printf("%u\n", &c);
```

Since only 1 byte of storage is allocated to a char variable in most C compilers intended for MS-DOS machines (and for that matter, the great majority of all microcomputers), the address returned names this single byte.

This sample program uses a char variable. However, such an application in a practical C program is rarely seen. Due to the conversions that take place in C, all chars are eventually converted to int types, just as all floats are converted to doubles. A single char variable is seldom used, even by functions that, for instance, capture a single character from the keyboard. One that comes to mind is getchar(). This function temporarily halts execution until a single character is input via the keyboard and followed by a carriage return. The getch() function (in most implementations) does the same thing, but restarts the execution chain immediately after any key is pressed. While both of these functions are made available for the intent of returning single characters, char variables should not be used as the return

variable. The reason for this is that both of these functions will return a -1 value if an error occurs. By convention, a char variable is an unsigned type (although the ANSI standard now makes provision for signed and unsigned char data types). An unsigned variable is not capable of storing a signed number as is the case with -1.

The most common use of the char data type is in the char array, an array of characters that represents a string. A string is often treated as a single unit in many computer languages, and it can be handled in this same manner in C. However, C does not have a true string variable. All strings are stored as an array of individual units. These units are characters or chars, and each character in an array consumes 1 byte of storage in most implementations.

The following program demonstrates a very common use of the char array.

```
#include <stdio.h>
#include <string.h>
void main(void)
{

     char c[10];
     strcpy(c, "COMPUTERS");
     printf("%s\n", c);

}
```

This program will display "COMPUTERS" on the monitor screen.

The invisible events are more numerous in this program. First of all, variable *c* is declared a char array allocated 10 consecutive storage bytes. Here, the programmer has direct control over how much storage space is set aside, whereas with all previous variables, the storage space was automatically set. As with the single char variable, storage is automatically set at 1 byte per char unit, but here ten 1-byte array elements are specified in the declaration line. Therefore, 10 *SSUs* (standard storage units) are allocated.

The reason for the selection of an array subscript of ten is due to the fact that the string constant "COMPUTERS" is to be copied into the array. Now, this constant contains only nine characters. The extra byte of storage is not just a safeguard. It is absolutely essential and not an extra byte at all.

The only element that makes COMPUTERS a true string in C and not just a consecutive trail of single characters is the operation of, in this program, the strcpy() function. This function is used to copy the characters that make up this constant to the memory positions set aside exclusively for array *c*. In C, the definition of a character string is "a single unit of characters terminated by a NULL character." The NULL character is ASCII 0. You don't see this character anywhere in the program, but the compiler automatically adds it to the end of the constant, COMPUTERS. The strcpy() function also copies the NULL character

into the tenth array byte. Therefore, the copy of the constant is written to the 10 consecutive bytes reserved for *c* as shown below.

```
COMPUTERS\0
```

The '\0' is the NULL character and signals the end of the character string. The NULL character makes this combination of single characters a single C string, a unit that may be treated as a single entity instead of a grouping of individual characters.

Don't be confused into thinking that strcpy() purposely tacks on an extra '\0' to the end of the quoted string used as its argument when it copies this string into the reserved memory locations. While this could easily be arranged by writing a new version of strcpy(), it is not necessary. This function is designed to terminate upon receiving a NULL byte but only after the NULL is copied to the new memory location allocated to, in this case, array *c*.

Remember that all constants written into a program that is to be compiled must be written somewhere in memory when the compiled program is actually executed. When the compiler sees the constant "COMPUTERS" in the source code, it writes it somewhere with the executable code that it produces. When this program is executed, this string constant is written to a safe place somewhere in memory. It is stored as

```
COMPUTERS\0
```

The compiler actually tacks on the NULL character, although it is never seen in the source code. Now, when the strcpy() function is invoked, it is handed the memory address of this constant and copies the contents from the constant's memory location into the bytes reserved for *c*. This includes the NULL character. As a matter of fact, encountering the NULL character is a signal to strcpy() to stop reading further bytes of information. When strcpy() has completed its run, there are two COMPUTERS strings in memory. One is the original constant that we will say, for purposes of discussion, is stored in the bytes that begin at memory address 61000. The second string is the copy produced by strcpy() at, again for discussion purposes only, memory address 64115.

Proceeding further, when printf() is handed a control string argument of %s, this is a signal that it is to expect a pointer argument. In this example, *c*, used without the braces, is that pointer argument. Printf() goes to that memory address and starts reading a string. But it does this a single character at a time. It writes each byte to the screen as an ASCII character. It will do this until it intercepts the NULL byte (\0). When this occurs, printf() stops reading this location and either exits or goes to another argument if the format string indicates that another is available. In this example, there are no other arguments so printf() is exited.

The first question that comes to mind is, "Where did printf() get a pointer argument?" This is a valid question, since there are no ampersands and no specifically declared pointers in this program. The answer lies in how C handles char arrays. The construct *c[0]*, for in-

stance, is a bona fide variable. It contains or returns the character in the first position of array *c*. However, when *c* is used without a subscript, it then becomes a pointer to the address of the first character in the string.

A char array name, when used without the subscript brackets, is a pointer to the start of the array storage and returns the memory location of the first byte, which is also the start of the string. It does not return the contents of that byte. It returns the byte's address in memory!

It can be safely said that *c[0]* is a variable. It can be reassigned any legal object value, as shown below.

```
c[0] = 66;
```

This reassigns the first character in our string the letter B. If this line were inserted into our program between strcpy() and printf(), then BOMPUTERS would be displayed on the screen. However, *c*, used without subscript brackets is a pointer. It returns not the contents of a byte but the memory address of the start of the string.

The address-of operator cannot be incorporated as shown below.

```
&c
```

The reason for this lies in the fact that *c* is already a pointer. However, it can be used with a variable.

```
&c[0]
```

This is perfectly legal. Remember, *c[0]* is a variable and *&c[0]* returns the memory address of this variable, the first byte in the array. This also happens to be the same address returned by *c*, the pointer to the start of the array. By the same token, *&c[1]* returns the address of the second byte in the array. It is a pointer to the address of this second byte.

The former program example can also be written in a different manner.

```
#include <stdlib.h>
void main(void)
{

     char c[10];
     int i;
     c[0] = 'C';
     c[1] = 'O';
     c[2] = 'M';
     c[3] = 'P';
     c[4] = 'U';
```

```
        c[5] = 'T';
        c[6] = 'E';
        c[7] = 'R';
        c[8] = 'S';
        c[9] = '\0'
        i = 0;
        while (c[i] != '\0')
            putchar(c[i++];

}
```

This program does exactly what the previous one did, but it replaces strcpy() with direct assignment lines. The *while* loop provides a very rough picture of how printf() works when displaying a string on the screen. Within the loop clause, termination instructions are given. The loop will cycle as long as *c[i]* is not equal to NULL. Note that *c[9]* is assigned a NULL value, which equates to decimal 0. Putchar() is used to display a single character on the screen during each pass. On each of these passes, variable *i* is incremented by 1. When *i* is finally incremented to a value of 9, *c[i]* is equal to 0 (NULL) and the loop is exited.

The analogy this latter program bears to the first one breaks down upon close examination. In the latter example, no direct pointer operations were used. In the former, pointers are everywhere.

DON'T ever think that, in the original program, *c* returns the string. This is totally untrue. It should be understood that C language doesn't really have any true string variables. However, C devises a way in which functions may access a series of characters until a stopping point is reached. This stop is signaled by the NULL character. Again, *c* does not equal COMPUTERS. It only returns the memory address at which the first character in this string is stored. Functions that accept this pointer are programmed to be *smart* and know to quit what they are doing after reading the NULL character. The next program will help demonstrate this point.

```
#include <stdio.h>
#include <string.h>
void main(void)
{

     char c[10];

     strcpy(c, "COMPUTERS");

     printf("%u\n", c);
     printf("%s\n", c);
     printf("%s\n", &c[0]);

}
```

When executed, this program displayed the following on the monitor screen.

```
65508
COMPUTERS
COMPUTERS
```

The first value is the memory address of the start of the array and will vary depending upon the type of machine and compiler used. Regardless, the address is that of the start of the array. The next line displays COMPUTERS. This occurs because the %s conversion specification handed to printf() causes the function to look for a memory address and then to display the contents at this address, consecutively and continuously until the NULL character is read. Next, COMPUTERS is displayed again. But why?

Earlier, it was stated that *c* is a pointer that returns the address of the first element in the array. It was also stated that *c[0]* is a variable and *&c[0]* is a pointer to that variable's memory location. Since this is the first variable in the array, its address is the same as the starting address of the array. In the last printf() line, a %s conversion specification was the only argument in the format string. Therefore, a pointer or memory address was expected. This criterion is met by *&c[0]*, and the full contents of the array are displayed. It can be readily observed that *c*, the pointer, and *&c[0]*, the pointer, point to the same memory address.

This occurrence is not unusual. C language had this feature written in by its designers. Using the array name without the subscript is just a shorthand method of stating the same thing with *&c[0]*. This is easier on the programmer, since less keyboard effort is required, but it sometimes tends to confuse beginners. If you want to have some fun, try to guess what the following program displays.

```
#include <stdio.h>
#include <string.h>
void main(void)
{

     char x[10];

     strcpy(x, "COMPUTERS");

     printf("%s\n", &c[3]);

}
```

The answer is PUTERS. Since the argument to printf() is the memory address of the fourth character in the array (remember, we start counting array elements at 0, not 1), the byte read by printf() begins at the letter P. As stated before, printf() is only given a starting point in

memory to begin reading byte contents. This instruction is provided by the %s conversion specification. The function reads these contents until a NULL character is encountered. Since the read began at P and the NULL occurs after the S, only this portion of the original string constant that was copied into the memory location reserved for array *c* is displayed.

It is important to state again that, in this context, *c* is a pointer and does not equal any object value (at least in the way we think of objects as opposed to addresses). It returns only the address of a memory location where objects have been written. It certainly does not equal COMPUTERS or even the first letter of this string. It is the equivalent of *&c[0]*, which is also a pointer. Both *c* and *&c[0]* point to the same memory location.

It should also be made clear that the array is in no way equal to the constant "COMPUTERS". Rather, this array contains an exact copy of the bytes that make up this constant. The constant that was an argument to strcpy() lies at one place in memory, and the copy that was made by strcpy() lies at another. This is extremely important in understanding pointer operations that will be discussed a bit later. Strcpy() is an aptly named function. It *copies* consecutive bytes of data. If a copy exists, then there must be an original. The constant is this original and the contents of the array are the copy. Both exist simultaneously in computer memory and at different memory addresses. This is not a figurative statement but a true-life fact.

Array Bounds Checking

One of the touted weaknesses of the C programming language is the lack of array bounds checking. This is the cost of relatively frugal use of memory, a strong point of C. Lack of bounds checking simply means that there are no safeguards that prevent an array from being overwritten. If a char array is declared with a subscript of 10, then only 10 sequential bytes of memory are reserved for storage to this array. However, if a programmer miscalculates and writes more than 10 characters (including the NULL) to this array, the excess characters are written into the memory locations that immediately follow the block that was allocated for the array. The 10 - element boundary of the array is passed, and bytes are written into memory locations not set aside exclusively for this array. When this occurs, what happens?

There is no single answer to this question. If there are few declared variables in the program, then there is a good chance that the unreserved portion of memory that was overwritten by the offending string is not being used anyway. The program may run in a normal fashion. However, if there are many variables, there is a very good chance that the exclusive storage allocated to these others may be overwritten by the long string. This will certainly cause an improper execution sequence. A worst-case scenario might involve those extra string elements overwriting a management portion of memory or even interacting with other variables to do the same thing. The result here can be a crash, when the computer simply locks up and must be re-booted. However, there have been cases where hard disk drives have been erased by inadvertent overwrites to interrupt addresses, a disaster!

Overwriting an array is the exact equivalent of writing a program that simply pokes random values into random memory locations. This is a crap shoot and anything goes! Obviously, this situation can bring about possible disasters such as the hard disk example. Programmers must be aware at all times of the boundaries associated with arrays and with the total size of any data that may be written to them.

The following program illustrates what can happen in an array overwrite.

```
#include <stdio.h>
#include <string.h>
void main(void)
{

     char a[3], b[10];

     strcpy(a, "LANGUAGE");  /* Overwrite */
     strcpy(b, "COMPILER");

     printf("%s\n", a);
     printf("%s\n", b);
}
```

The expectation here is the following display.

```
LANGUAGE
COMPILER
```

However, the strcpy() assignment to array *a* is an overwrite! Three array elements were reserved for storage by the following assignment.

```
char a[3];
```

However, LANGUAGE consists of 8 characters and will consume 8 bytes of storage. We must also take into account the NULL character (\0) for a total of 9 bytes. The array bounds are exceeded, and there will be no error message or warning when the program compiles. The Borland C++ compiler used for testing all applications in this book yielded the following screen display. Most other types of compilers will be affected in the same manner.

```
LANGCOMPILER
COMPILER
```

The reason for this aberration lies in the way storage is managed. Let's assume that array *a* was allocated 3 bytes of storage at memory location 65504 in the 64K segment. Array *b* was

then allocated 10 storage bytes beginning at 65508. Here is what happened. LANGUAGE was written in memory starting at 65504. Because 9 storage bytes were required to contain this string, bytes 65504 through 65510 were utilized. But, allocated storage for array *b* began at 65508, so the supposedly exclusive storage allocation for this latter array was invaded! However, LANGUAGE was still written to memory. This would have been fine if the assignment to array *b* had not been made. When the second strcpy() function was executed, it began writing COMPILER at address 65508. Therefore, this last string was written atop the portion of the first string that intruded into its exclusive storage area.

Such an overwrite was not in any way disastrous in this simple program. The error was immediately recognizable and could have been be easily corrected by allocating more storage space to array *a*. However, if this program had been more complex, an overwrite of this nature could have lead to many tedious hours of debugging.

This simple example should provide a very worthwhile lesson in array handling. This lesson will carry over (but magnified 100-fold) when dealing with declared pointers.

The Little Man Method

In the fictitious neighborhood discussed in the last chapter, there is a small town composed of many little houses in which reside little men with knapsacks on their backs. Some knapsacks have many compartments for storing bytes. Some have only a few. The little man that represents a char variable has a very small knapsack. It contains only a single compartment for the storage of 1 byte. The following program will continue this story.

```
/* CHARACTERS */
#include <stdio.h>
void main(void)
{

     char c;

     c = 'A';

     printf("%u\n", &c);
     printf("%c\n", c);

}
```

As before, we known the little man is named *c*, but we don't know where he lives. To find his home address, it is necessary to use the PRINTF electronic directory at the corner of CHARACTERS Street. The ampersand pointer, when aimed at *c*'s name, returns the address where he lives.

On this same street, there is a townhouse complex where all of the units are all tied together but are also separate from each other. This domicile will provide a storybook example of the char array, which is demonstrated by the following program.

```
#include <stdio.h>
#include <string.h>
void main(void)
{

     char c[9];

     strcpy(c, "COMPILER");
     printf("%u\n", c);
     printf("%s\n", c);
}
```

In this townhouse development, there are nine units. In each of the first eight units, one little man resides. All of these little men are brothers, and all have the same last name, which happens to be *c*. Since we know the last name, the POINTER on the street corner directory is aimed at *c*. (Remember, *c* without the brackets is a pointer. The ampersand is unnecessary.) This gives us the address of the first townhouse in the complex, 65504 CHARACTER Street. The first-born brother lives in the first townhouse (*&c[0]*), the second-born in townhouse number two (*&c[1]*), etc. Remember, there are only eight brothers but nine townhouses in this complex. The last townhouse (*&c[8]*) is vacant. The last printf() line in the above program represents a special feature that is a part of this townhouse directory. It will allow us to see the contents of each brother's knapsack. When this directory is implemented, the contents of each knapsack are displayed in sequential order. The contents are sampled as long as there are knapsacks available. However, when the ninth townhouse is reached in this poll, it is empty. There is no little man and, of course, no knapsack. The special directory scan is halted.

Incidentally, if we want to know the address of an individual brother, we would utilize the standard ampersand pointer that is also available at this street corner directory and point to the desired brother's name. Their names are *c[0]*, *c[1]*, *c[2]*, *c[3]*, *c[4]*, *c[5]*, *c[6]*, and *c[7]*. Therefore, placing the ampersand in front of the proper brother's name will yield his address. Placing the display feature of this directory on an individual brother's name (no ampersand here) reveals the contents of this individual's knapsack.

We will forgo further formal use of the Little Man Method, although he will appear now and then in future discussions. It is hoped that this storybook approach has enabled any reader who may have had conceptual difficulties to get the correct feel of the operations under study.

Summary

It can be seen that, unlike single-element variables, char arrays offer more sampling variations. We can find the address of the start of the array by using the array name that is a pointer to the start of array storage as an argument to a printf() function that contains the %u conversion specifier. We can also display the entire contents of this array by using its name again, still a pointer, as an argument to a printf() function that contains a %s conversion specifier. The address of any single element in the array can be obtained by using the address-of operator in front of the array name followed by its bracketed subscript. Here, it will be necessary to use the %u specifier again. If we merely want to return the object stored by a single array element, the element name without the ampersand is used with a %c specifier (for the display of a single ASCII character).

For all intents and purposes, each element in a char array may be treated exactly like a standard variable of type int, except that only 1 byte of storage is allocated instead of 2. But, the entire contents or combined contents of the array may also be accessed as a single unit. This provides great flexibility, uncommon to many other languages, and should not be very confusing if you are aware of the two-part nature of character arrays. The first part is that of a group of individual values. The second is the string that rolls all of these values into a single unit.

Chapter 4

Declared Pointers and Their Operations

A pointer is a special variable that returns the address of a memory location. This statement has been repeated again and again. There is nothing especially mysterious about pointers and what they do, but initial concepts often breed their own mysteries. This may be the real problem with programmers and pointers.

The discussions so far have expanded on some very basic operations in ANSI C. The reader has gained an added or, at least, a reinforced insight into the memory management that takes place in even the simplest programs. These insights will be a great aid in garnering the most from the discussions that are to follow.

All of the pointers that have been discussed at this juncture have been fixed, that is, they have been inextricably tied to variables that were declared at the opening of the program. This chapter deals with variables that are declared from the onset to be pointers. These variables do not store objects, at least as we think of them in regard to other variables. They store an address of a memory location. Unlike the previous pointers discussed, these can be made to point to any area of memory.

The following program begins the exploration into this area of declared pointers in C language programs.

```
#include <stdio.h>
void main(void)
{

     int i, *p;

     i = 43;
     p = &i;
     printf("%d  %d\n", i, *p);

}
```

The declaration line of this program names *i* a variable of type int. The same line also declares that *p* is a pointer. The *indirection* operator (*) is the key here. The declaration indicates that the combination of this operator and variable *p* is an int data type. This, then, is a pointer that is structured to point to an address in memory that contains or can contain a legal int value.

I realize that this discussion may be very confusing, but bear with me. Just remember for now that *i* is an int variable and *p* is a declared pointer of type int.

Next in the program, int variable *i* is assigned a value of 43 decimal. The statement that follows this assigns the pointer (*p*) the address of the start of storage allocated to *i*. After these assignments have been made, variable *i* contains an object of 43, and *p* points to the first byte of memory allocated to *i*.

When the printf() function is called, it is handed two arguments. Both of these are decimal integers, as indicated by the %d conversion specifiers used in this function's format control clause. The screen will display the following.

```
43  43
```

These two values are one and the same. The printf() function got these two arguments from the same memory location. When the declarations were made, variable *i* was allocated 2 bytes of storage for a standard integer. Pointer *p* was allocated only enough memory to contain a memory address. This is all a pointer can do. In a standard variable, we have learned to expect an object and a memory address. In a pointer, the object is a memory address, and like any other object, its value can be changed. Here is a special variable that seems to have the capability of moving through memory. This is true, based upon appearances, and for the time being we will embrace this conceptualization.

As important as the ability to move around in memory can be, using the indirection

operator, we can retrieve bytes of data from wherever the pointer is pointing or even write new data to that area of memory.

The simple program under discussion usually leads to more questions than it answers, but it does serve as a suitable starting place. Now, how is the previous program different from the one that follows?

```
#include <stdio.h>
void main(void)
{
      int i, p;

      i = p = 43;

      printf("%d  %d\n", i, p);

}
```

This program uses two conventional int variables and causes the two copies of the constant (43) to be stored in memory. One of these 43s is stored at the memory address set aside for exclusive storage to variable *i*. Another copy is made at the memory address set aside for variable *p*. There are no declared pointers in this program.

The important difference is that this program requires storage space for two int values (43). Therefore, a value of 43 exists twice in memory, each being separate objects assigned to separate variables. In the former example, there was only a single value of 43. It was assigned to variable *i*. Pointer *p* was assigned the memory address of variable *i*. When the printf() function was called, it displayed the same value of 43, the first having been assigned as the object to *i*, the second being the object that resided at the memory location contained in pointer *p*.

The on-screen display using either program will be identical, but the source of the values actually displayed is different. The former program contains a single object of 43 stored at a single memory location. The latter example contains 43 at two memory locations. (Both programs also contain another value of 43. This is the constant that was written directly into the program.)

We have just seen how a declared pointer may be used to return the object value from the memory location to which it is directed to point. The following example shows how the pointer may be utilized to write a value to the same memory location.

```
#include <stdio.h>
void main(void)
{

      int i, *p;
```

```
        i = 43;
        p = &i;
        *p = 16;
        printf("%d\n", i);

}
```

When this program is compiled and executed, it will display a value of 16 on the screen. Note that variable *i* is assigned a value of 43 in this program, but when it is passed as an argument to printf(), this value has been changed. Variable *i* now has an object value of 16. The reason for this lies in the pointer operation.

After the declaration line, assignments are made. Variable *i* is assigned a value of 43, and the pointer, *p*, is assigned the memory address of this variable. This means that *p* points to the exclusive storage set aside for variable *i*. However, another assignment line uses the indirection operator in connection with the pointer. This combination is the object value at the memory location. **p* accesses the byte contents at the memory address. Now, these contents are overwritten with a value of 16. This means that the value of 43 is overwritten and in its place is the value of 16. Since this is the area of storage exclusively allocated to *i*, the original object value in *i* has changed. This occurs even though no reassignments were made directly to the *i* variable.

Throughout the discussion of declared pointers, the pointer has always been assigned the address of a declared int variable by using the address-of operator (&) in conjunction with the variable name. We have already learned that this combination returns the memory address of the storage set aside for the variable. However, if the specific memory location of this variable is already known, the address assignment to the pointer can be made directly and numerically. The following program illustrates this. To be fully understood, you must assume that the storage address of int variable *i* is 65514. Just how this came about is not important, relevant, or possibly even accurate. But, for the sake of discussion, we will assume this is the correct address of variable *i* upon declaration.

```
#include <stdio.h>
void main(void)
{
        int i, *p;

        p = (int *) 65514;
        *p = 78;

        printf("%d\n", i);

}
```

When this program is executed, the object value in *i* will be displayed on the screen and this value will be 78. This was assigned to the variable indirectly by simply writing it to the memory location allocated for *i* and to which *p* points. Again, we are assuming for the sake of this discussion that 65514 was the location the compiler allocated for storage to *i*. You will observe that when the pointer is assigned a memory address value, a cast operator is used in the form of (int *). This operator casts or coerces the numeric value (an unsigned integer constant) into an int pointer type. Some compilers will issue warnings if this cast operator is not used, and many will consider it an error not to include it.

Be aware that the following assignment

```
p = (int *) 65514;
```

is exactly equivalent to

```
p = &i;
```

if we assume that the memory address of variable *i* is at 65514 in the 64K data segment.

A Practical Use of Pointers

The program examples to this point have been of very little practical value other than as simple tutorials to teach the basics of pointer operations. This discussion delves into some practical uses of pointers, utilizing them in constructs where no other type of variable will serve nearly so well.

Every beginning C programmer at one time or another has tried to do what is illustrated in the following program:

```
#include <stdio.h>
void change(int i);
void main(void)
{

      int x;

      x = 146;

      change(x);

      printf("%d\n", x);

}
```

```
void change(int x)
{

      x = 351;

}
```

This program is very simplistic and absolutely incorrect. The attempt here is to change the object value of a variable from within a customized (programmer-written) function. This simply can't be done. When this program is executed, it will display the value of 146 on the screen. The operations that take place within the function have no bearing whatsoever on the object value of the variable that was passed to change().

In the C programming language, arguments are passed to functions *by value*. This simply means that the value in variable *x* in the example above (146) is passed to the change() function. The function variable, also named *x*, is a discrete entity, having no relationship with the variable from the calling program.

Just as all arguments to functions are passed by value, all a C function can do is to return a value to the calling program. It cannot change the object value by acting on the value of a standard variable. However, pointers allow us to make the in-memory changes attempted by the previous program. Using pointers, the program could be correctly written as follows.

```
#include <stdio.h>
void change(int *i)
void main(void)
{

      int x;

      x = 146;

      change(&x);

      printf("%d\n", x);

}
void change(int *x)
{

      *x = 351;


}
```

This will work as originally intended. The value displayed on the screen will be 351. The program is set up as before, but the change() function is passed the address of variable *x*, instead of its object value. Within the definition of the change() function, the passed value is declared a pointer of type int by the following line.

```
void change(int *x)
```

Now, the object value in this memory location, *x, is overwritten by the value of 351. The function has been able to locate the exclusive storage area for variable *x* in the calling program, change it, and then relinquish control to the calling program. Incidentally, the change() function used the name *x* as its pointer variable just as the former, incorrect, example did. This is certainly not mandatory or even desirable. The change() function could just as easily have been written as follows.

```
void change(int *i)
{

      *i = 351;

}
```

A function is a completely separate program from that managed under the auspices of main(). The variables, pointers or otherwise, within a function are not the same variables that are contained in the calling program, even though they may have the same names. In this last program, the change() function actually creates another pointer of type int that is assigned the same memory address value as that of the pointer value that was passed from the calling program. There are two pointers, one under main() and one under change(). Both point to the same memory location. A change in object value stored at this single location has the same effect, regardless of which pointer was used to make the change. The pointer with the function was used to change the value at the memory address allocated to *x* in the main program. The variable in the calling program is used by printf() to display the object value.

Note that the memory address of the variable in the calling program is passed as &x, using the address-of operator to obtain the pointer value. In the function, the pointer is simply declared in the standard fashion using the indirection operator. The argument is passed to the function as a fixed pointer (fixed to the variable), whereas the pointer variable within the function is declared a pointer from the start.

Perhaps the best illustration of the use of a pointer to make object changes to the calling program from within a function is the swap() function, first discussed in K&R's *The C Programming Language*. An example of this is illustrated in the following program.

```
#include <stdio.h>
void swap(int *i, int *ii);
void main(void)
{

      int x, y;

      x = 10;
      y = 127;

      swap(&x, &y);

      printf("%d  %d\n", x, y);

}
void swap(int *a, int *b)
{

      int i;

      i = *a;
      *a = *b;
      *b = i;

}
```

Again, addresses are passed from the calling program to the swap() function. These arguments are declared int pointers within the function, which also declares an internal variable of type int. The latter is a standard auto variable and not a pointer. Variable *i* is used to temporarily store the object value in **a*. Remember, when used in this context, **a* is the object value of the bytes stored at the memory location pointed to by *a*. The indirection operator dictates this type of return. Next, the object at the memory address pointed to by pointer *a* is reassigned the object value that resides in the memory location held by pointer *b*. Again, the indirection operators are used to access the objects at the memory addresses as opposed to the address, proper. Finally, the bytes pointed to by *b* are reassigned the value in *i*. The swap is complete and the program will display the following on the monitor.

```
127  10
```

This occurs because the object values of *x* and *y* in the calling program have been swapped.

DANGER!!!

Some of the dangers of pointer operations were alluded to in earlier chapters. Since pointers can be directed to any area of memory, one must make certain such an area is not used for other purposes of which the programmer may have no knowledge. In the last few program examples, declared pointers were always given the address of existing variables as shown below.

```
p = &i
```

Here, *p* is a declared pointer and *i* is a declared auto variable. Since storage was specifically allocated to *i* upon its declaration, it's safe for the pointer to point to this memory address. As such, the pointer can only be used to retrieve or to write the object value in *i*. The following program demonstrates a very dangerous use of a pointer and indicates what things programmers inexperienced with pointers sometimes attempt.

```
/* Danger!! Don't execute this program */
#include <stdio.h>
void main(void)
{
      int *x;

      *x = 10;

      printf("%d\n", *x);

}
```

Here, a pointer of type int is declared followed by an assignment to a value of 10. Remember, a pointer always points to a memory location. When a pointer is used to write an object to memory, as is the case here, we must know where in memory that assignment is made. The big question in this example is, "To what or where does *x* point?" The answer is that it's anybody's guess!

We learned that when an auto variable (of type int for this discussion) is declared, memory is set aside for its exclusive storage. We also learned that the bytes at this storage location are not cleared to zero and may contain any value on a more or less random basis. The same is true of pointers.

However, the object value of a pointer is a memory address. Therefore, when a pointer is declared, the address it contains, the area of memory to which it points, can be any random value of from 0 to 65535 (assuming use of a small-memory model compiler using

2-byte pointers). Here, again, is the proverbial crap shoot. If you have ever written and run a program along the lines of this latter example, you were writing things to memory locations not reserved for that purpose. You were POKEing around in memory at random.

There's a pretty good chance that, if your program was simple and small, like this example, nothing unusual took place and the program ran as expected. However, all this really means is that you got lucky! There is also the possibility that your machine locked up and had to be re-booted or even that another part of your program was trashed.

When objects are written to random memory locations, anything can happen. This sort of programming error is exactly equivalent to the array overwrite discussed in Chapter 3, and must be avoided at all costs.

Other Numeric Pointer Types

The examples of declared pointer operations discussed here have all been of type int. This simply means that the declared pointer expects to point to a memory location that contains or will contain a value within the range of a standard integer. The previous swap() function allowed the values of two int variables in the calling program to be exchanged. However, this same function would not work if handed the addresses of double-precision floating-point variables.

In the C programming language, a pointer can be of many different data types. The following program shows how the previous swap() function needs to be altered in order to exchange the values of two double variables.

```
#include <stdio.h>
void swap(double *d, double *e);
void main(void)
{

      double x, y;

      x = 14.673;
      y = 0.39712349;

      swap(&x, &y);

      printf("%lf  %lf\n", x, y);

}
void swap(double *a, double *b)
```

```
{
        double i;

        i = *a;
        *a = *b;
        *b = i;

}
```

The only change in the function is that the pointer variables have been declared double instead of int. The internally declared variable *i* is also changed to a double-precision floating-point type. Everything else remains the same. Of course, the arguments passed to swap() are the addresses of double variables declared in the calling program.

While it is proper for the programmer to be thinking in terms of memory addresses when dealing with pointers, it is also as important to know what kind of object is to be written or returned via the pointers. One would no more expect to read from or assign a double value to a memory location set aside for an int value than to assign the object value of a double directly to an int variable.

Void Pointers

The void data type is a null value. This type was first introduced with ANSI C, as it was not a part of the original C language. Any function that does not produce a useful (or needed) return is usually declared to be of data type void.

In pre-ANSI C, pointers of type char were the generic pointer types because of their 1-byte access alignment. Functions such as malloc() and calloc() in this earlier version of the language typically returned pointers of type char. In ANSI C, the void pointer type has the same address alignment requirements as a char pointer. Therefore, void pointers are now often used in place of the earlier and less expressive char pointer types where applicable.

Prior to the introduction of void, a pointer of type char was usually incorporated to reference some part of memory without having to be concerned with the type of data stored there. For example, the original version of malloc(), the memory allocation function, returned a char pointer. Under ANSI C, malloc() returns a pointer of type void.

The main concept behind a void pointer is that of a pointer that can be used conveniently to access any type of object. This avoids much of the typecasting that would be required if the pointer were of another type such as char. The following program makes use of a void pointer.

```
#include <stdio.h>
void main(void)
```

```
{

        void *v;
        char *c;

        c = "Testing void pointer";
        v = c;

        printf("%s\n", v);

}
```

This program will display the following string.

```
Testing void pointer
```

The address of the string constant is assigned to char *c. Next, the address in the char pointer is assigned to the void pointer *v. Finally, the void pointer is handed to the printf() function, which uses the %s conversion specifier to display a string value. The end result is exactly the same as would be the case if the char pointer were handed the same function.

This proves that a void pointer in ANSI C can be used to point to any data type. It can be assigned any address from any other type of pointer and does not require casting for such operations to take place. A void pointer type in ANSI C truly is a generic pointer.

Summary

This chapter has dealt with only the basics of declaring pointers of numeric data types. In each case, the declared pointers were used to point to the location of the same type of auto variable. This is known as *initializing a pointer*, causing it to point to a known memory location.

When a pointer is declared, it is uninitialized, just as is the case with a declared auto variable. Therefore, an uninitialized pointer has a random object value, the object being a memory address. An uninitialized pointer, then, can point to any area of memory. This makes it a potentially dangerous item that can wreak havoc with a program.

Remember, a pointer is declared for the specific purpose of storing the address of a known memory location. Such a location may be specified by the programmer in two different ways. The direct method entails constructs similar to the one that follows.

```
x = (int *) 65117;
```

In this example, 65117 was arbitrarily chosen. It is assumed that the programmer has a specific need or desire to access this location. The indirect method is illustrated by the following statement.

```
x = &y;
```

Here, *x* is a pointer and *y* is a declared variable with automatic storage set aside for its objects.

Both of these methods return a numeric address to the pointer. The first is specified as a constant, while the second retrieves an address by using the address-of operator preceding the variable name. We don't necessarily know the specific address using the latter method, but the pointer does and that's what's important.

Pointers are very useful as arguments to functions that must access memory locations within the calling program. Remember, arguments are passed to functions by value. Without pointers, functions could do no more than pass values back to the calling program. With pointers, functions can change values in memory that affect the calling program with no further actions on the part of the function.

The use of the indirection operator with a pointer name causes the object value at the memory address stored in the pointer to be returned. Also, the object value may be changed by making a new assignment via the pointer name and the indirection operator. When used with this operator, the pointer may be treated exactly like an auto variable of the same type as that under which the pointer was initially declared.

It must be stressed that a pointer really holds nothing with the exception of a memory address. This is its object value. It does not hold a string, it does not hold an integer, it does not hold a double. In this regard, it doesn't hold anything. It points to a place in memory, and through the use of proper operators, the object values at the address to which it points can be retrieved or written.

A declared pointer is not fixed to point to any single memory location. It can be directed to rove throughout the entire quantity of RAM. The examples of pointers discussed so far assume the use of a small-memory model compiler. Using MS-DOS, the small-memory models are confined to a 64K segment of memory for data storage. As such, they cannot be made to point to locations outside of this segment. The reason for this lies in the fact that small compiler pointers are allocated only 2 bytes to store a memory address. Within 2 bytes, the maximum unsigned value is 65535. Today, even low-level MS-DOS machines are usually equipped with at least 1 megabyte of memory and most contain 2 megabytes or more. The latter configuration offers memory addresses more than thirty times beyond the range of the 64K limitation placed on small model compilers.

Other discussions in this book will bring into play large model compilers, but the reader should understand that the small models are most desirable since they are the most efficient. This means that they produce compiled programs with the smallest object code size and run faster than those compiled with large models. Sometimes it is necessary to forgo these qualities for the all-memory access feature of the large models. As a rule, however, use the small-memory model compilers wherever possible.

Chapter 5

String Pointers

In ANSI C, a pointer may be declared as any legal data type including char. The char pointer is an often-used instrument for accessing and returning character strings. For most purposes, char arrays and char pointers may be treated identically, especially in regard to being passed as arguments to functions. During the earlier discussion, it was learned that the name of a char array (without the use of the subscripting brackets) is indeed a pointer to the start of that array. In C, arrays of characters may be treated as strings by terminating the series of characters with a NULL (\0). The NULL is what identifies a character string as opposed to a sequential order of individual characters. The NULL is used for purposes of reading the string as a single unit.

The following program demonstrates the declaration of a char pointer and its potential use.

```
#include <stdio.h>
#include <string.h>
void main(void)
{

      char a[40], *p;

      strcpy(a, "DATA");

      p = a;

      printf("%s\n", p);

}
```

This program will display the string "DATA" on the monitor screen, because *p* points to the start of the 40 sequential bytes of memory allocated to array *a*. As with previous pointer declarations, the indirection operator is used when declaring a char pointer. The strcpy() function copies a string constant into the bytes allocated to the array. Next, *p* is assigned the address of the array. When printf() receives the argument, it is a pointer to a string in memory.

Note that the address-of operator (&) was not used with the array in making the address assignment to pointer *p*. Remember, *a* is the name of the array and is also a pointer itself when used without the brackets. Therefore, *a* returns an address to *p*, which is expecting one. While totally unnecessary, the pointer assignment line could have been written as follows.

```
p = &a[0];
```

This would assign *p* the same address as before, since it is the address returned by the original assignment line. You won't see an assignment like the latter in most ANSI C programs, as it is redundant.

The previous program displays the string pointed to by *p* when this pointer is used as an argument to printf(). We know that *p*, the pointer, has exactly the same significance as *a*, the array name, which is also a pointer because of the lack of brackets. We also know that the following program statement will return the first character in the string to printf() where it will be displayed as a single character.

```
printf("%c\n", a[0]);
```

To do the same thing with the pointer, the following program would suffice.

```
#include <stdio.h>
#include <string.h>
void main(void)
{

      char a[40], *p;

      strcpy(a, "DATA");

      p = a;

      printf("%c\n", *p);

}
```

We need an object value as an argument to printf() when the %c conversion specifier is part of the format string. The designation *a[0]* is an object value and so is **p*, which returns the object value at the single memory location pointed to by *p*.

In each of these examples, the declared pointer is always initialized by making it point to the memory location of a declared array. We can say that the pointer points to a safe area of memory that is composed of 40 consecutive bytes, since this is the subscript value of the array. However, both of the previous programs can be made much simpler and will consume less memory by resorting to the following programming mode.

```
#include <stdio.h>
void main(void)
{

      char *p;

      p = "DATA";

      printf("%s\n", p);

}
```

This program does exactly the same thing as the first example in this chapter, but it does not require the declaration of a char array. It does not use the strcpy() function, and it is simpler to write. From a memory standpoint, we are dealing with only a single variable, and that one is a pointer.

"But, you assigned a pointer an object value and the pointer wasn't initialized!" That may have been the first reaction the reader had to this program, and it would seem that we

have done just that. After all, *p* wasn't made to point to a char array declared in the program. Or was it?

This program is perfectly correct, but it can lead to confusion since it seems to go against everything that has been previously stated about the use of pointers. Let's answer the question regarding initializing a pointer by giving it the address of a specific area of memory instead of the random value to which it points when first declared. It has been previously stated that C does not make any great provisions for strings. The only difference between a series of separate characters and a string is that the string is terminated by the NULL character. This is exactly what occurs when a string constant is included in a C program.

This program uses the constant "DATA", which is surrounded by quotation marks, usually referred to as double quotes in C jargon. This constant must be stored at a memory location when the program is executed, and it is to this location that *p* is made to point by the following assignment line.

```
p = "DATA"
```

This doesn't assign to *p* the object value of "DATA", only the address of the memory location where this constant has been stored. Therefore, *p* does point to a safe area of memory, the area where the constant is stored. Most programmers can grasp the concept of memory being set aside exclusively for declared variables, but memory is also set aside to store constants. Since a pointer may be directed to point to different areas of memory, it is just as easy to make it point to an area used to store a constant as it is to an area of memory allocated for use by a variable.

This sample program is only slightly different from the one that follows.

```
#include <stdio.h>
void main(void)
{

      printf("%s\n", "DATA");

}
```

We already know that the %s conversion specification tells printf() to expect a char pointer argument, a pointer to the start of a string in memory. The objects at this memory location will be read and displayed on the screen as single characters until the NULL is encountered. Therefore, "DATA" is a pointer. In the earlier program example, *p* will contain the same address as "DATA". If you don't fully understand this, the following program may help.

```
#include <stdio.h>
void main(void)
{

      printf("%u\n", "DATA");

}
```

This program will display the memory address where the first letter in the string constant is stored. On the model system used for researching this book, the address is at memory location 162 in the 64K data segment. This storage location will change slightly by also declaring a char pointer, as in the following program.

```
#include <stdio.h>
void main(void)
{

      char *p;

      p = "DATA";

      printf("%u\n", p);

}
```

On the model system, address 158 is displayed, which means that "DATA" was stored at locations 158, 159, 160, 161, and the NULL character at 162. One might think that this could be confirmed by printing the address of *p* and the address of "DATA" in the same printf() line. The following program is an example of how one might try to accomplish this goal, but it won't work!

```
#include <stdio.h>
void main(void)
{

      char *p;

      p = "DATA";

      printf("%u  %u\n", p, "DATA");

}
```

This program doesn't perform in the manner many programmers may expect. The idea

here is that two identical memory addresses will be displayed on the screen. No way! The two addresses will be different. Why?

The reason is the fact that two different constants are used in this program. The first use of "DATA" is not equal to the second use of "DATA". Although both constants are identical in their character makeup, they are two discrete constants, each stored at a different location. Pointer *p* has been assigned the memory address of the first constant. The address of the second constant is stored elsewhere. Therefore, the address in *p* is different from the address of the constant used as an argument to printf(). The address in *p* is the same as that of the first constant.

The char pointer in ANSI C is a valuable aid to the programmer in many endeavors that include more than displaying or accessing string data. Such pointers are often required as arguments to functions that perform mass PEEK/POKE operations as well as other forms of memory management.

Unlike the int pointer that can be used to read or write a 2-byte quantity, char pointers address areas of memory a single byte at a time. A char variable is allocated 1 byte, and a char array with a subscript of 40 allocates 40 single and consecutive bytes of data for its exclusive storage. Prior to ANSI C, a standard C function such as malloc() would often return an address to a char pointer for that pointer to be used for single-byte access. With the advent of ANSI C, the void pointer type is more often called upon for the same purposes.

Returning to earlier statements regarding the treatment of char pointers in the same manner as char arrays, study the following program example.

```
#include <stdio.h>
#include <string.h>
void main(void)
{

      char a[5];
      int x;

      strcpy(a, "DATA");

      for (x = 0; x <= 4; ++x)
            printf("%c\n", a[x]);

}
```

This program will display a vertical string of characters on the monitor.

```
D
A
T
A
```

It does this by accessing each character in the array and treating it as a single entity. The bracketed subscript is advanced by the stepping of the loop so that characters in array positions 0-4 are read independently by each loop cycle. Here, the string value is unimportant (as a string, proper). The idea is to access each character on an individual basis.

This same operation can be done more efficiently by using a char pointer. The pointer equivalent of the previous program follows.

```
#include <stdio.h>
void main(void)
{

      char *a;
      int x;

      a = "DATA";

      for (x = 0; x <= 4; ++x)
            printf("%c\n", *(a + x));

}
```

The key to this operation lies in the use of the indirection operator in conjunction with the pointer value added to the loop variable value. If *a* is a pointer with the address of a string, then *a + 1* is the address of that same string plus 1. For instance, if the address in *a* is 128, then *a + 1* is equal to 129, or memory address 129. Taking this further, if *a* is the address of the start of a string, then *a + 1* is the address one character into the string. It should be stressed that this incrementation takes place in SSUs. Since the storage unit for a char data type is 1 byte, incrementing a char pointer by (numerical) 1 will increment the pointer address by 1 byte (1 SSU for char types). However, if the pointer is an int type using an SSU of 2 bytes, then incrementing this pointer by (numerical) 1 would result in the pointer address being incremented by 2 bytes (1 SSU for int types).

Likewise, if **a* returns the first character in a string, then **(a + 1)* returns the second character in the string. The parentheses are necessary to establish order of precedence from a

purely mathematical standpoint. It can be seen that **(a + 3)* is the same as *b[3]*, assuming that *b* is a char array whose address is contained in pointer *a*.

Now, why is the second program more efficient than the first, since they both do the same thing? The answer lies in the use of memory or rather in the lack of memory use in the second program. The first example declares a char array with 5 elements. This means that 5 bytes must be set aside for exclusive storage to this array. Next, a string constant is used that also requires 5 storage bytes. The contents of the constant are copied to the array elements. Once this procedure is complete, there are two separate strings containing "DATA" residing in program memory, a total of 10 bytes for program string data.

In the second example using the char pointer, we still have the string constant, requiring 5 bytes of storage, but we do not have to reserve another 5 bytes of storage for an array to which this constant can be copied. There is no char array. Therefore, the string data storage requirements of this program are half that of the previous program. Of course, the pointer itself requires 2 bytes of storage, but this still represents a savings of 3 bytes.

Big deal! A lousy 3 bytes! So what? These are good observations, but percentages will be more revealing. The latter program was 30 percent more efficient from the standpoint of string data storage. If we were dealing with 1000 strings, a 30 percent savings is nothing at which prudent programmers will turn up their noses. This is sometimes difficult to comprehend when such simple program examples are used for tutorial purposes. Just remember that ANSI C programs may require thousands, hundreds of thousands, and even millions of storage bytes, depending upon the scope and complexity of a particular program. Any reasonable degree of memory conservation could be essential.

There is one other area of efficiency in which the latter program excels. There is no strcpy() function as in the first example. This function is unnecessary because no copy of the constant is made. The pointer is directed to the memory location of the constant. Since the function does not have to be invoked, the storage required for its source code is not a part of the executable program. More memory space is spared. Plus, the execution time required by the strcpy() function is saved. The end result is that the latter program executes faster and requires considerably less memory space. This is a sizable savings that will multiply by the complexity of any programs that require similar programming structures.

Using a typical small-memory model C compiler, the first program that used a char array resulted in an object code size of 265 bytes compared with 237 bytes for the pointer version. When ANSI C applications are executed under MS-DOS, there is always a high overhead contained in the operating system interface that allows the program to be supported and executed. The executable (.EXE) files for these two programs measure out at 5778 and 5730 bytes, respectively, with the pointer version being the smaller of the two. This represents an approximate 11 percent memory savings regarding the object code and less than 1 percent savings in executable code size for the pointer version. However, as an ANSI C program builds in complexity, the overhead tends to remain at relatively the same size. A much greater savings would be had if many, many operations were to be carried out when

comparing the pointer version to the char array example.

It must be understood that the following expression does not change the value of the address in *a*.

```
*(a + 3)
```

Rather, it simply uses this initial address as a reference point, adding a value of 3 to the base address to access a particular character. However, there is a slightly simpler method of accomplishing the same thing the previous programs did that will save a bit more storage space. The following program demonstrates this method.

```
#include <stdio.h>
void main(void)
{

      char *a;

      a = "DATA";

      while (*a != '\0')
                  printf("%c\n", *a++);

}
```

Such an alteration reduces our object code to 233 bytes. This method actually changes the value of the pointer. Within the *while* loop the following exit clause is used.

```
*a != '\0'
```

This simply states that the loop is to continue cycling until the character returned by **a* is the NULL, which is represented in standard ANSI C syntax as '\0'. This is the character equivalent of numeric 0. Within the printf() function, the following expression is found.

```
*a++
```

This is the standard usage of the incremental operator, which may be used in front of the variable name or following it. When the incremental operator precedes the variable name, the variable value (memory address, in this instance) is incremented, and then the object value is returned. When it follows the variable name, the object value is returned, then the address is incremented. The latter operation is what is desired in this program. Each time printf() is executed within the loop, the current object value or character is returned by **a*; then the memory address in *a* is incremented by 1 SSU. If we assume that "DATA" begins

at address 158, then the D is located at this address. After the D is returned within printf(), the address contained in the pointer is incremented by 1 SSU. On the next pass of the loop, *a* is pointing to 159. The A is returned to printf() and the address is incremented again. This continues until the NULL character is read, signalling the end of the string. The loop starts to cycle again, but the exit clause is satisfied. At this point, **a* is equal to '\0', and the loop terminates.

Notice that the method of stepping the value of the pointer eliminates the need for the int variable used in preceding programs. This saves storage space. Again, the amount in these examples is negligible, but in larger programs, the savings that can be had by this type of programming technique can make a very large difference.

There is an even more compact way of performing the same operation carried out by the previous program examples.

```
#include <stdio.h>
void main(void)
{

      char *a;

      a = "DATA";

      while (*a)
                  printf("%c\n", *a++);

}
```

This program does not save any additional storage space when compared to the previous example, but it is quicker to write because all extraneous source coding is eliminated. The change occurs in the escape clause within the *while* loop. This notation may seem strange, but that's part of the beauty (and the cause of headaches for beginners) of the ANSI C syntax.

The escape clause within the *while* loop is merely a conditional test.

```
*a != '\0'
```

This clause returns a value other than 0 as long as the character in **a* is not equal to numeric 0. Again, the character '\0' and the number 0 are the same. So the *while* statement is going to continue looping as long as its escape clause does not equate to 0. When **a* does equal 0, the clause also equates to 0, meeting the criteria of the escape condition, and the loop is terminated. In this example, the fact that the clause returns a 0 when **a* returns a 0 is just a coincidence. If the escape clause in while() had read

```
while (*a != 65)
```

the clause would equate to 0 when the first A in "DATA" was returned. ('A' is equal to ASCII 65.) The clause equates to 0 because the conditional test is not met. This condition states that **a* is not equal to 65. The clause returns a TRUE value of other than 0 as long as the condition of the test is met. However, when **a* is indeed equal to 65, the condition is no longer met, and a FALSE value or 0 is returned.

When manipulating string quantities, we are always checking for the end of the string, which is numeric 0. We can use this to advantage when writing ANSI C programs that access strings. This is exactly what has been done with the last program example.

We know that the last character in a string is '\0', or numeric 0. So instead of going to the trouble of writing an escape sequence that evaluates the return value of a clause, let's simply evaluate the true value of the pointer object. Therefore, *while* evaluates the content in **a*. As long as this value is not 0, the loop continues to cycle. When the NULL character is read, the numeric value is 0 and the loop is exited. Again, *while* will continue to cycle as long as it reads a value other than 0.

It is important to remember that these last examples have incremented the address stored in the pointer. When the loop is exited, the pointer no longer has the address of the start of the constant; it contains the address of the NULL character at the end of the constant string. This can be proven by the following program.

```
#include <stdio.h>
void main(void)
{

      char *a;

      a = "DATA";

      printf("%u\n", a);

      while (*a)
                  printf("%c\n", *a++);

      printf("%u\n", a);

}
```

On the model machine used for researching this book, this program will display the following.

```
158
D
A
T
A
162
```

The breakdown on storage of the string constant is shown below.

```
 D    A    T    A   \0
158  159  160  161  162
```

The pointer that was initially given the address of 158 now contains an address value of 162. The latter value is where the NULL character is stored. Programmers must remember that when the address value in a pointer is incremented, its object value, which is the initial address, is changed just as the object value of an int variable is changed when it is incremented.

This condition makes no difference to this program since it terminates right after the loop is exited. Also, when a pointer is handed as an argument to a function, the variable that represents this function argument is not the same pointer that was handed to the function. Rather, it is another declared pointer that contains the same address as the one in the function argument.

The pointer that is incremented from within a function does change memory address value. However, this in no way changes the address value of the pointer from the calling program. Remember, arguments are passed to functions by value. A pointer passes the value of the address it contains. This value is assigned to the pointer declared within the function. Any changes in the initial value passed to the function are in effect only within the function body. Values passed from calling programs are never altered. This applies to pointers as well as to auto variables. However, when a function has a memory address, it can write changes into that address. The following program further explains this concept.

```
#include <stdio.h>
void vert(char *c);
void main(void)
{

      char *a;

      a = "DATA";

      printf("%u\n", a);

      vert(a);

      printf("%u\n", a);

}
void vert(char *p)
{

      while (*p)
```

```
        printf("%c\n", *p++);

}
```

This program displays the following.

```
158
D
A
T
A
158
```

This means that "DATA" is stored at relative memory address 158 and that *a* points to this address before entering the vert() function. The address in *a* is passed to this function as a value that is then assigned as the memory address of *p*, the char pointer that is declared within the function.

The print operation takes place from within the function body by incrementing the address value in *p*. When the function is exited, *p* contains the address value of 162, the NULL character position in the constant's storage area.

However, the address value in *a*, the argument pointer, remains unchanged. The major emphasis here is that *a* is not passed to the function! The memory address value IN *a* is passed again, as a value. The memory address is that of "DATA", and that location is accessed by *p*, which initially points to the same thing as *a*.

Problems

It's not uncommon to see a program similar to the one that follows.

```
#include <stdio.h>
#include <string.h>
void main(void)
{

      char *a;

      strcpy(a, "DATA");

      printf("%s\n", a);

}
```

This is absolutely wrong. To what does *a* point? You don't know the answer to that ques-

tion and neither do I. This is a prime example of overwriting memory. When *char *a* is declared, it points to a random location in memory. In computer jargon, we can say that *a* points to garbage as soon as it is declared. However, one programmer's garbage may be another's treasure. As was stated in prior chapters, when you overwrite computer memory, you're involved in a game of logic Russian roulette, and the outcome can be almost anything. The program may work or seem to work perfectly. If this is true, then the hammer fell on an empty chamber. The computer could have locked up, or worse, returned inaccurate information. Bingo! The hammer fell on a loaded chamber. While such occurrences are rarely disastrous when using small-memory model compilers, large-memory model versions may allow a memory overwrite to trip a few interrupts and, maybe, erase the file allocation table (FAT) on your hard drive. Double bingo! You've just rendered your entire hard disk useless!

Another common and incorrect use of a char pointer is demonstrated in the following program.

```
#include <stdio.h>
void main(void)
{

      char *a;

      gets(a);

      printf("%s\n", a);

}
```

Here again is the notorious memory overwrite. The gets() function retrieves characters from the keyboard and stores them in memory. The key word here is *store.* Store means that there must be room to put something somewhere. Where in memory are the keyboard characters to be put? They will be stored in the series of memory locations beginning at the address pointed to by *a.* Where does *a* point? Anywhere!

A pointer must always be initialized by assigning it a memory address! This applies in every case. The address can be specified directly, as a numeric value, or as is more often the case, it can be named indirectly by assigning the declared pointer the address of a variable or other program object.

Any time an attempt is made to copy something to a pointer, a mistake is being made. A pointer can only be assigned a memory value. This breaks down into a practice of "the pointer equals this" with "this" being an address.

All of the previous incorrect examples of pointer usage tried to copy bytes of data into unknown memory locations. The proper type of variable to be used for copying char string

data is a char array, sized to meet the maximum string input from the keyboard. It's important to remember that once a char pointer is given the address of a char array, the pointer points to a safe area of memory for storage of up to the maximum amount of characters specified in the array subscript. The following program demonstrates this concept.

```
#include <stdio.h>
void main(void)
{
      char a[40], *p;

      p = a;

      gets(p);

      printf("%s  %s\n", p, a);

}
```

This program will accept a keyboard input of up to 39 characters (plus the NULL) safely; the array can store 40 bytes without overwriting memory. The pointer, *p*, is given the address of the array, which means that it points to 40 safe bytes of memory. When this program is executed, a keyboard input of

```
COMPUTER <Enter>
```

will result in

```
COMPUTER   COMPUTER
```

being written to the display screen. This occurs because the pointer serves as the argument to gets(). The bytes input at the keyboard are written to the storage area to which *p* points. Since this is the area allocated for *a*, the string is copied into this array. Both *a* and *p* will return the same character string to printf().

Of course, the program would be more efficient without the pointer (in this particular example) and would best be written as follows.

```
#include <stdio.h>
void main(void)
{
      char a[40];
```

```
        gets(a);

        printf("%s  %s\n", a, a);

}
```

This example writes the same information to the screen, and works in exactly the same manner, since in the previous example, both *p* and *a* pointed to the same memory location.

Can you deduce what the following program will display on the monitor screen?

```
#include <stdio.h>
#include <string.h>
void main(void)
{

        char a[40], *p;

        strcpy(a, "SILICON");

        p = a;

        strcpy(a, "CONTROL");

        printf("%s\n", p);

}
```

If you answered SILICON, then you are incorrect. The correct answer is CONTROL. Remember, the pointer, *p*, was given the address of the array. It was not made to point to either of the two string constants, only to the start of array storage. When *p* is assigned the address of *a*, the pointer immediately points to the first string copied into *a*. The bytes that make up SILICON were stored in *a* at the time of this initial assignment. However, the next program operation copies a new string into *a*. CONTROL overwrites SILICON. The pointer still points to the memory address of *a*; therefore, *p* now points to CONTROL, which resides at the same location previously used to store SILICON.

When a pointer points to the address of an array, this means that the pointer is completely interchangeable with the array as far as arguments to functions such as printf() are concerned. Some would say that *p* is equal to anything *a* is equal to. This is correct but can also be misleading. Since both *p* and *a* are pointers (remember, the array name becomes a pointer when used without the subscripting brackets), they both contain the same memory address. It is more accurate to say that when a declared pointer is given the address of an array, the contents of the array and the memory content that the pointer accesses are always

the same.

There is a difference, a big one, in a variable being *equal* to a value and in one being the *same* as that value. The following program demonstrates this difference.

```
#include <string.h>
void main(void)
{

      char a[40], *p;

      strcpy(a, "GARAGE");

      p = "HOUSE";

}
```

Hopefully, this will clear up any misconceptions surrounding the terms *equal to* and *same as*. Here, the string constant "GARAGE" is copied into the char array. From an earlier discussion, we know that this causes the bytes that make up the constant to be copied or reproduced in the array. When this copy operation is complete, there are two strings in memory. The first is the constant "GARAGE". The second is the copy of the constant, also "GARAGE". Therefore, the contents of the array are equal to "GARAGE", but they are not the same as the constant "GARAGE". The constant resides at one memory location, the contents of the array at another.

On the other hand, the pointer is assigned the memory address of the constant "HOUSE". No copy takes place. "HOUSE" is found at only one location in program memory. Therefore, *p* points to "HOUSE", the constant. What *p* points to is the constant itself. In this regard, they are the same. The address of "HOUSE" and the address to which *p* points are the same.

In human terms, we can compare this example with identical twins. They look alike. You can't tell the difference in the two when they stand side by side. However, there are two of them. They are not one and the same. In the above example, "GARAGE" was twinned. By the same token, "HOUSE" is an individual. Only one exists. Therefore, anything that equates to "HOUSE" is "HOUSE" and not a copy.

Arrays of Pointers

At this juncture, we have compared char arrays to char pointers. An array is a collection of single data units, while a pointer, as always, is a special variable that holds the address of a memory location. However, arrays of pointers are not only possible but practical as well. The following program will start this portion of the discussion.

```
#include <stdio.h>
#include <string.h>
void main(void)
{

      char a[5][15];
      int x;

      strcpy(a[0], "DATA");
      strcpy(a[1], "LOGIC");
      strcpy(a[2], "MICROPROCESSOR");
      strcpy(a[3], "COMPUTER");
      strcpy(a[4], "DISKETTE");

      for (x = 0; x < 5; ++x)
            printf("%s\n", a[x]);

}
```

This program simply copies five string constants to the two-dimensional array, *a*. In making the declaration, the second portion of the subscript determines maximum string length. Since it is known that "MICROPROCESSOR" will be the longest string and that it contains 14 characters, the minimum string length declared for the array is 15 characters or bytes. This will be just large enough to contain the 14 characters in "MICROPROCESSOR" plus one more (15 total) for the NULL character.

The multidimensional array can also be classified as an array of strings or even as an array of char arrays. In any event, the array declaration does not allow dimensioning for different string lengths. If the maximum string length is 15, then storage of this quantity is provided in all 5 of the subscripts. This applies even though most of the strings are fewer than 8 characters in length. Since 15 bytes are provided for each string, there is an obvious waste of storage. The strcpy() function is used to make a copy of the string constants and to store each in the bytes for each string in the array.

Finally, a *for* loop is entered that causes the contents of the array to be written to the screen. This is a multidimensional array, so the expression *a[x]* is a pointer to the *x* element or *x* string in the array. This may seem to conflict with earlier discussions where it was stated that the array variable name without the subscript brackets was a pointer, while the array name with brackets returned the object. This applies only to single-dimension arrays. With two dimensions, a single set of brackets with the name is a pointer, while the following construct returns the object at the array position specified by the values of *x* and *y*.

```
a[x][y]
```

Therefore, the expression

```
a[0]
```

where *a* is a multidimensional array, is a pointer to the first element of the first string in the array. *a[1]* is a pointer to the first element of the second string, etc.

In addition to the wasted memory usurped by the array that must size all array elements to the length of the maximum expected string, there is also the factor of storage being required for constants as well as for copies of those constants. This is a most unsatisfactory condition and should be remedied, if possible. Pointers make this a distinct and easily accomplished possibility as is demonstrated by the following program.

```
#include <stdio.h>
void main(void)
{

      char *p[5];
      int x;

      p[0] = "DATA";
      p[1] = "LOGIC";
      p[2] = "MICROPROCESSOR";
      p[3] = "COMPUTER";
      p[4] = "DISKETTE";

      for (x = 0; x < 5; ++x)
                printf("%s\n", p[x]);

}
```

This program declares an array of char pointers. The following designation declares *p* to be an array of five char pointers.

```
char *p[5];
```

The pointer to the first string is *p[0]*, the second string *p[1]*, the third *p[2]*, etc. As always, nothing is copied to a pointer. Rather, it is assigned a memory address. In this case, each of the five pointers in the array is assigned the address of a constant. This avoids the copying procedure from the previous program and the doubled storage requirements. The rest of the program is handled in a fashion that is identical to the previous example. The execution run appears to be identical to the one that used the multidimensional char array.

However, from a memory usage standpoint, this program is very different. Using a typical ANSI C compiler on the test computer, the multidimensional char array version consumed 409 bytes in object code format and 5880 bytes as an executable file. The pointer version of this same program required only 331 bytes for the object code and 5800 bytes for the executable file. This equates to a 23 percent savings in code size for the object module and about 1.5 percent decrease in executable code. This is still not highly significant, but the percentages of savings seem to grow as program complexity increases, especially in regard to the storage of objects.

Summary

C language pointers of type char are the most common of all. It can be seen that char pointers are closely aligned with char arrays, and the two can be used interchangeably in many applications. At the same time, there are still very significant differences between the two, mostly regarding storage techniques. Like all variables, char arrays are allocated storage space based upon the subscript value specified by the programmer. Char arrays are used to store data. However, like all pointers, those of type char are allocated no storage areas for common objects (in this case, char data). The only thing a char pointer can store is a memory address.

Upon declaration, a char pointer has the address of a random memory location. This uninitialized address value should be considered garbage, useless for any practical programming purpose.

It is only when the char pointer is assigned an address that it becomes initialized. It then points to a place in memory, again, specified directly or indirectly by the programmer. Any variable, pointer types included, is useless and dangerous when it contains a random value. It is always necessary to give the pointer something at which to point. False assumptions about the address a pointer contains can lead to, at minimum, a faulty program that is very difficult to debug. In a worst-case scenario, data may be written to a protected memory location, creating havoc with the operating system interfaces and possibly resulting in a permanent loss of stored data.

A char pointer expects to point to a memory location allocated to store data of type char. This equates to one byte of storage for one character in most MS-DOS systems. Char pointers are often used as an efficient means of manipulating string constants. They may also access the storage areas of char arrays. In the latter usage, they may be treated just like arrays. The important concept here is that the char pointer must always contain the memory address of a safe area of memory, one designed for char or char string operations.

Chapter 6

POINTERS AND MEMORY ACCESS

Since pointers have the ability to point to different areas of memory and to read and write information at these locations, it would seem obvious that they would be the ideal instruments for performing manipulations in and to memory. Every computer language contains some mechanism that will allow a memory location to be located for the purposes of returning the byte content from that location or to alter the byte in that location. Such mechanisms are usually referred to as peek/poke devices.

All of the discussions that have been offered to this point have used program examples that were compiled using the small-memory model version of the test compiler. Such models restrict memory access to a 64K data segment unless special *far* pointers or functions are used.

This chapter explores the large-memory model of the ANSI C compiler as well as far pointers. However, the start of this discussion still uses the small-memory model, as it is adequate for introduction purposes.

The following program is a partial repeat of several others from Chapter 5, but it is examined in a different light.

```
#include <stdio.h>
void main(void)
{

      char *a;


      a = "ANGULAR";


      printf("%d\n", *a);


}
```

Note that the printf() function uses a conversion specification of %d instead of %c. The latter would display the value in **a* as a character. The %d specification causes this same value to be displayed as a decimal integer. The value displayed on the screen will be 65, the ASCII code for 'A'.

We know that pointer *a* is given the address of the constant in the following assignment line.

```
a = "ANGULAR";
```

Let's assume for the purposes of this discussion that the storage location for the constant begins at address 158 in the 64K segment. What has this program done?

The answer is that it has *peeked* at memory location 158 in the segment. The byte value at this address is 65. This clearly demonstrates that all a peek consists of is returning the byte value from a specified memory location.

The original C programming language contained a peek() function, and all popular versions contain similar functions in various forms and under the guise of various names. However, a peek function in C is rather redundant since pointers allow for such easy access to this operation by direct programming methods.

The following program demonstrates a *poke* operation.

```
#include <stdio.h>
void main(void)
{

      char *a;

      a = "ANGULAR";
```

```
        *a = 66;

        printf("%s\n", a);

}
```

When this program is executed, it will display

```
BNGULAR
```

The following assignment line within the program causes the byte referenced by *a* to be replaced (overwritten) by the assignment to **a*.

```
*a = 66;
```

Remember, the indirection operator causes the specific byte to be returned or, in this case, written by assignment. What has occurred here is a typical poke operation. Pointer *a* points to the memory address where the constant is written, specifically, to the first byte in the string at location 158. **a* returns this byte or can be used to write another byte on top of it. In this case the new byte has a value of 66 and is poked into memory on top of the initial value of 65. ASCII 65 is the letter 'A' and ASCII 66 is the letter 'B'. Memory has been altered by this poke operation. The printf() function is used to display the result of the change made in memory, thus, "BNGULAR" is displayed.

Memory Models

Throughout this book references have been made to the various memory models available in ANSI C compiler packages. Most modern ANSI C compilers for MS-DOS machines offer several categories of memory models. These can be classified into two basic categories, small- and large-memory models. A specific compiler may have four or more memory model options that are often classified as tiny, small, and compact, each of which falls into the overall classification of small-memory models. Large-memory models are broken down into medium, large, and huge. For our discussion, the main difference between these two memory model classes is in the default size of the pointers they directly support. The small model, which has been used for all programming examples to this point, has 2-byte pointers, while large-memory models offer 4-byte pointers.

From an execution efficiency standpoint, the small-memory model offers the fastest execution speed and smallest code size. However, the small model also limits its pointers to 2-byte entities. A 2-byte pointer contains the same storage capacity of an unsigned integer. (Note: Byte size for various types of data may vary on different types of computers. Most

MS-DOS C implementations utilize the same storage parameters as does the model compiler used in researching this book, i.e., 1 byte for chars and voids, 2 bytes for ints and unsigned numbers, 4 bytes for floats and long integers, and 8 bytes for double-precision floating-point values.)

An unsigned integer can store a maximum value of 65535 (64K). Any pointer that is declared in the normal fashion using a small-memory model compiler is allocated 2 bytes for address storage. This applies regardless of the type of pointer that is declared. Since only 2 bytes of storage are allocated, this means that the maximum memory address value is limited to 65535. This severely limits the memory that can be addressed, as a typical system may have addresses spreading out to twenty times this amount or far more.

Early (pre-ANSI) C compilers for MS-DOS machines were available only in small-memory models (for the most part). Most made no provisions for memory excursions outside of the 64K data segment. Later versions offered special functions that would address any valid memory address. Newer compilers were offered in large- and small-memory models, and the latest compiler versions also allow for the use of special 4-byte pointers to be specifically declared from within small-memory models.

A programmer who wants the fastest possible execution speed will usually try to stay within the confines of the small-memory compiler versions. Prior to the newest type of compiler, the need to rove around in all of a computer's memory forced the use of the large-memory models and their slower execution speeds and larger code sizes. Today, it is possible to use a small-memory model and specifically declare 4-byte or far pointers when memory excursions outside of the 64K data segment are desired.

For specific peek/poke operations, most modern ANSI C compilers also offer functions that address these. Borland C and C++ compilers use functions named peekb() and pokeb(). The first returns a single byte from memory based upon segment and offset arguments. The second writes a single byte to memory based upon the same arguments plus the byte value to be written. Other brands of ANSI C compilers may offer similar functions, although they may be given different names. Such functions are really nonstandard and are made a part of compilers for the convenience of users. Bear in mind that such functions will most likely not be portable.

Since many programmers who are new to ANSI C come from the ranks of BASIC programmers, the simple program that follows is presented. This code was written in Microsoft BASIC and writes the letter 'C' directly to the monochrome screen buffer.

```
10 DEF SEG = &HB000
20 POKE 0, 67
```

The letter 'C' is equal to ASCII 67, which is the value used with the POKE statement. It will be written in the upper left hand corner of the screen. Since BASIC works in a manner similar to a small C compiler version, only one 64K area of memory can be accessed at a time. The DEF SEG statement allows the address of any 64K segment to be defined. The

POKE statement can access any of 65535 bytes in this segment. We can move through all of memory with this combination, since if we need to go outside of the specified 64K segment, we simply change the DEF SEG value. In the above example, &HB000 is the address of the monochrome screen buffer specified in the customary hexadecimal format. This HEX value is equivalent to decimal 45046. Programmers who are using a VGA monitor must specify an address of &HB800 (47104 decimal) which is the start of the color screen buffer (CGA EGA VGA). The POKE statement's first argument is the offset into the segment. In this case the offset is 0, which targets the first byte at &HB000. The second argument is the byte to be written to the specified memory location. ASCII 67 is the letter 'C'.

Using the Borland compiler, the pokeb() function allows for the same type of addressing as was demonstrated by the previous BASIC program. To accomplish the same thing using the small-memory model ANSI C compiler, the following program can be used.

```
#include <dos.h>
void main(void)
{

      pokeb(0xb000, 0, 67);

}
```

Here, the DEF SEG value from the previous BASIC program serves as the first argument to the function. The second argument is the offset into the segment and the third argument is the byte value to be written. The &H designation in BASIC means that the number to follow is in hexadecimal format. The Hex format is indicated in ANSI C using the 0x (zero-x) prefix.

However, it seems rather wasteful to use a function to perform this simple operation, especially since we already know that pointers can address any area of memory. The following program must be compiled using the large-memory model of the ANSI C compiler.

```
void main(void)
{

      char *a;

      a = (char *) 0xb0000000;

      *a = 67;

}
```

That's all there is to it. No function has to be invoked that will result in slower execution.

The poke operation is handled directly by the char pointer. The only unusual feature of this program to some readers will be the address value. One must remember that far pointers, the type created when any pointer is declared in the normal fashion using a large-memory model compiler, are 4-byte entities. With 8 bits to a byte, we can say that 2-byte pointers require 16-bit numbers and 4-byte pointers must have 32-bit addresses. This is what the latter is, a 32-bit address specifically naming the monochrome screen buffer segment of memory in absolute terms. In 8-bit terms, we can break the 32-bit address down into a segment and an offset.

```
              SEG      OFFSET
0x      b000          0000
```

If you are accustomed to providing addresses in 8-bit values, and most users of MS-DOS machines and software are, then simply follow the hex value with four zeros. Purists will opt for the mathematical method where the 8-bit address by is multiplied by 0x10000. If you prefer to work within the decimal system (cumbersome when dealing with memory addresses), then multiply the decimal memory address by 65535 decimal.

In any event, the previous program must use absolute addressing when dealing with memory locations, and each address is given as a 32-bit (4-byte) quantity. 0xb0000000 is the address of the start of the monochrome display buffer. (VGA monitor users will use 0xb8000000 to access the color screen buffer.) The assignment line uses a cast operator to coerce the numeric value to type char *. This is necessary in type casting the numeric value to a form that is acceptable to the pointer.

Once a memory address has been assigned, we know that *a* points to the start of, in this case, the monochrome screen buffer. This means that **a* will access the object value stored in the byte at this location. Therefore, the following construct reassigns a value of 67 to this byte.

```
*a = 67;
```

The letter 'C', which is represented by decimal 67, appears in the upper left corner of the screen. This is far more efficient, both from a programming and an execution speed standpoint when compared to calling the pokeb() function. Calling a function from a program requires additional execution time.

When a C program calls a function, the calling program actually relinquishes control to the function. This takes more time than if the function were actually written within the calling program, proper. You can literally see the difference calling a function like pokeb() makes in execution speed when a slow XT-class computer is used. The following program represents a method of filling the monochrome screen with 'C' characters using pokeb().

```
#include <dos.h>
void main(void)
{

      int x;

      for (x = 0; x <= 3999; x += 2)
                pokeb(0xb000, x, 67);

}
```

This program should be compiled under the large-memory model compiler option. This is necessary for comparison purposes with the program that is to follow. Before moving on to the comparison program, let's discuss a few traits of this one.

The monochrome or color text screen is comprised of 4000 single bytes. Each even byte (including 0) will display a character on the screen when poked with an ASCII value. In this case it is 67 for the letter 'C'. However, each odd byte (1 to 3999) is called an *attribute* byte. The standard screen fills each attribute byte with a value of 7, as this will bring about normal display of each character. Other attribute values will cause the preceding character to be displayed in flashing, bold, or underlined format. Since the screen attribute bytes already contain a value of 7 for normal display, it is unnecessary to address the odd bytes at all in this program example.

The *for* loop steps variable *x* from a value of 0 to 3999 in increments of 2. On the first pass of the loop, *x* is equal to 0. On the next pass, it will be equal to 2, then 4, then 6, etc. This means that only the even bytes will be accessed. These are the bytes to which character ASCII codes are to be written.

The pokeb() function is called on each pass of the loop. Note that the segment value is always 0xb000, the 8-bit code for the start of the screen buffer. (Note: If you are using a color monitor, this value should be changed to 0xb800.) The pokeb() function requires an 8-bit address, even when it is compiled by the large-memory model compiler. The stepping value of *x* is used as the offset argument, while 67 is the byte to be poked into all accessed locations.

On the first pass of the loop, 67 will be poked into byte 0xb000 + 0. On the next pass, the byte location is 0xb000 + 2, then 0xb000 + 4, etc. When the loop times out, all of the character bytes in the screen buffer will be filled with the 'C' character.

Even when using a very slow computer, you will see these characters written very rapidly to the monitor screen. Those readers with ultra-fast computer clocks should try to toggle machine speed to the lowest setting possible in order to appreciate the time differences between this and the next program.

This program calls a function that, in turn, declares additional variables, executes its

code, and then returns control to the calling program. When the loop recycles, the function is called again, and the process is continued until the loop times out.

The next program does exactly the same thing (on-screen) as the former example, but it calls no function. Rather, it programs a pointer to perform the poke operation.

```
void main(void)
{

      char *a;
      int x;

      a = (char *) 0xb0000000;

      for (x = 0; x <= 3999; x += 2)
                *(a + x) = 67;

}
```

Compile this program using the large-memory model option as before. When it is executed, the monitor will be filled with a full screen of characters (seemingly) instantaneously. This method of poking in characters is much faster, because no function was called. For all intents and purposes, the pokeb() function works along the same principles that this program incorporates. With functions, however, there is a far greater overhead in that they must declare additional variables that this program avoided entirely. If you are using a fast computer, you won't be able to detect the speed difference in the two methods, but it is quite significant.

Newcomers to C language often become confused over the avant garde use of characters such as 'C', 'f', 'L', etc., interchanging them with numbers. C language makes no distinction between 'A' and the numeric value 65. They are both the same. Certainly, functions are available that will display 65 as 'A' or vice versa. The following simple integer assignment may be written in different ways.

```
x = 66;
```

The statement below means exactly the same thing.

```
x = 'B';
```

In both cases, variable *x* is assigned a value of 66 decimal. How this value is displayed on the screen depends upon the conversion specification given to printf(), assuming this is the function used for display. A conversion specification of %c will cause a numeric value argument of 66 to be displayed as the letter 'B'. If the conversion specification is %d, then the

same value will be displayed as decimal 66. If you go in for hexadecimal numbers, then a specification of %x will cause decimal 66 to be displayed as HEX 42. Similarly, a specification of %o will cause 66 to be displayed as octal 102. All of these values, 66D, 'B', 42H, 102-OCT, are equal. The numerical base in which they are displayed is up to the programmer.

Now that the mysteries of poking bytes into memory have been explained, let's concentrate on peek operations. The process is the same in that memory locations are accessed. However, instead of writing a byte of data to such a location, the byte already in that location is read. The following program demonstrates this operation using the Borland peekb() function.

```
#include <stdio.h>
#include <dos.h>
void main(void)
{

      int x;

      x = peekb(0xb000, 0);

      printf("%d\n", x);

}
```

The function allows the segment to be input as an 8-bit quantity, while its second argument is the offset (number of bytes) into that segment. This function returns a value of type int. The return is the object value of the byte at the location specified. The segment address is the start of the monochrome display buffer, so if the letter 'A' appears in the upper left corner of the screen prior to running this program, *x* will be equal to 65, the decimal equivalent of character 'A'.

The following program shows a more efficient method of accomplishing the same thing by directly programming a pointer to handle the access.

```
#include <stdio.h>
void main(void)
{

      char *a;

      a = (char *) 0xb0000000;

      printf("%d\n", *a);

}
```

This code must be compiled using the large-memory model option of the C compiler. As with the equivalent example of a poke operation, this program declares *a* to be a pointer of type char. This pointer is then given the address of the screen buffer in 32-bit format. Now, instead of assigning a value to **a*, we simply print the value already in **a*. The returned byte is displayed as an integer value, although it could also be displayed as a character, a hex value, or even an octal number if desired. What is contained in **a* is a single value. How it is displayed is up to you.

How would you go about writing your own function to handle a peek operation? True, the direct method is most efficient, but if you had to write such a function for some unknown purpose, you would simply follow the program logic presented above and end up with something similar to the following source code.

```
int peeker(long loc)
{

      char *mem;

      mem = (char *) loc;

      return(*mem);

}
```

That's all there is to it. Of course, this assumes that the address handed to the function (represented here by *loc*) would be a 32-bit quantity and that this function would be used only in programs compiled under large-memory models. A long int argument is handed to this function, since this variable's storage capabilities are adequate to handle full memory addressing. A char pointer is declared within the function and is assigned the value in *loc* as an address. Once our pointer is in the right location, all that is necessary is to return the byte.

If you would like a small-memory model version of this same function, it is first necessary to go into a little more detail about the features offered in modern C compilers. The advantage of the small-memory model options in most C compilers lies in the fact that 2-byte pointers are used. This facilitates execution speed and allows for smaller code size. The disadvantage of this model is that the same 2-byte pointers cannot be used to address all memory locations. This means that you are stuck with functions that will address memory for you along with the slower execution speeds that are incurred through their use.

The advantage of the large-memory model options of the same compilers is that the 4-byte pointers provide full access to all memory addresses. However, using large models will result in slower equivalent execution times and larger code.

One solution is to use a small-memory model compiler that offers the option of declaring 4-byte pointers. This is exactly what the best C compilers do today. The small-memory models may be used in the standard way and will provide compact code along with the

fastest execution speed. All pointers declared in the normal fashion will be 2-byte types. However, if you need a special pointer to access a memory location outside of the code area restricted by the 2-byte entities, then just declare that special pointer.

Such pointers are declared by using the *far* keyword. The following program is compiled using a small-memory model option and illustrates the difference between near and far pointers.

```
#include <stdio.h>
void main(void)
{

      char *a;
      char far *b;

      printf("%d   %d\n", sizeof(a), sizeof(b));

}
```

The first pointer, *a*, is declared using standard conventions and is allotted 2 bytes for storage. The second is modified by the *far* keyword and is a 4-byte entity. All of this assumes that the program is to be compiled under a small-memory model option. The printf() function is used to display the size of each pointer in bytes using the intrinsic C language sizeof() operator. When the program is run you will see that pointer *a* has a size of 2 bytes and *b* is sized at 4 bytes. Again, this program must be compiled using one of the small-memory model options.

Now we have the best of both worlds: small-memory model speed and economy and the ability to declare far pointers to address any area of memory. Note that the declarations of these pointers were made on two separate program lines, but this was done for the sake of clarity. The more obvious method of making these two declarations would be in one line.

```
char *a, far *b;
```

This declares *a* to be a standard (2-byte) pointer, while *b* is a far pointer, both of type char. Incidentally, the name *far* is quite appropriate and is based upon assembly language nomenclature of near and far operations. The far pointer is able to gain access far away from the data segment into other memory areas. The standard, 2-byte pointer is known as a near type, since it must stay nearby, within a predetermined memory segment due to its 2-byte storage limitation.

The following program is the equivalent of the peek operation discussed earlier, but it is compiled under one of the small-memory model options.

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

```
void main(void)
{

      char far *a;
      a = (char far *) 0xb0000000;

      printf("%d\n", *a);

}
```

That's all there is to it. You will notice that the cast operator contains the far modifier. You must remember that *a* is not a char pointer but a *char far* pointer. Notice also that the address is given as a 32-bit quantity as is required for 4-byte pointers. Other than the addition of *far*, the program is identical to a previous example and returns the first byte in the screen buffer.

Now, back to the problem that brought on this latest discussion in the first place. How does one write a peek function that can be compiled under a small-memory model, one that will accept 8-bit addresses as the peekb() function does? It's easy using the far pointer option available.

```
int peeker(int seg, int off)
{

      char far *mem;

      mem = (char far *) (seg * 65536);

      return(*(mem + off));

}
```

Again, there's nothing to it! Our new function, called peeker() for lack of a better name, accepts two arguments. Both are of type int, the first being the 8-bit segment address, while the second names the offset into memory.

The following designation simply adds the offset value to the segment value and returns the byte at that address.

```
*(mem + off)
```

This corresponds to the operation of peekb() that was discussed earlier. BASIC programmers can think of *seg* as the DEF SEG value and *off* as the single argument that would be

passed to the PEEK() function. Of course, we need to go from an 8-bit numeric quantity to the 32-bit equivalent because *mem* is a far pointer. This is accomplished by simply multiplying the memory address in *seg* by 65535 decimal, as this value is being assigned to the pointer. You could also express the multiplier value as 0x10000 to keep with the hexadecimal memory addressing convention.

Another method of writing this same function follows.

```
int peeker(int seg, int off)
{
      char far *mem;

      mem = (char far *) (seg << 16) + off;

      return(*mem);

}
```

The (possibly) questionable line is shown below.

```
mem = (char far *) (seg << 16) + off;
```

This is quite a mouthful, but it's easily explained when taken in small segments.

```
seg << 16
```

This portion uses the *left-shift operator* in C language. This means exactly the same thing as multiplying seg by 65535. This method requires less keyboard input and is more expressive when dealing with HEX values. After this operation takes place, the offset value is added on. Therefore, *mem* points to the proper address of *seg*, converted to a 32-bit address, plus *off*. This example is shown because C programmers often use shortcuts that are not as clear as some of the simpler examples found in this book.

Of course, this function is superfluous, as we can accomplish the same thing without resorting to a function as has been amply demonstrated. The direct method will result in the most efficient program.

Why Char Pointers?

At this juncture in my seminars and C language workshops, someone invariably asks why char pointers are used for these purposes and not int or long types. That's a good question, and there is a good answer.

A char pointer expects a return value of type char or can be used to write a value of type

char. (Remember, a char can be specified as the character itself as in 'C' or as its ASCII code, 67. Both mean the same thing.) A char value is a 1-byte entity. On the other hand, int pointers expect an int return, which is a 2-byte value on MS-DOS systems. A long or float is a 4-byte entity, and a double consumes 8 bytes. When peeking and poking around in memory, we normally wish to do this one byte at a time. Other pointer types will not allow for a 1-byte return or a 1-byte write, at least not in the direct manner char pointers do.

"But you said all pointers in small-memory model compilers for MS-DOS machines are 2-byte entities, and in large models, they have 4 bytes for storage!" That's right, but this applies only to the area set aside for the pointer to store an address! The number of bytes they return or access is fixed at 1 for chars, 2 for ints and unsigneds, 4 for floats and longs, and 8 for doubles (again, assuming a typical MS-DOS implementation).

The following program (compiled with a small-memory model) will help explain all of this.

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{

      int far *x;
      char far *a;

      x = (int far *) 0xb0000000;
      a = (char far *) 0xb0000000;

      system("cls");

      printf("%c\n\n\n", 'A');

      printf("%d\n%d\n", *a, *x);
 }
```

This program declares two far pointers. Pointer *x* is of type int, while *a* is of type char. Both pointers are given the address of the start of the monochrome display buffer. The system() function is used to call the CLS command, which clears the display screen and resets the cursor to the upper left hand corner. A printf() function is then called, which prints an 'A' in the upper left corner of the screen followed by two carriage returns, or *newlines* as they are known in C terminology. (Readers who are using a VGA monitor will want to change the address value to 0xb8000000.)

Another printf() function is used to display the values in **a* and in **x*. Remember, both of these pointers have been given the same address. Both point to the first byte of screen

memory, but *a* is of type char and *x* is an int type.

The expectation here is that the final printf() program statement will display two values of 65. But that's not what happens! Instead, you get 65 on one line and 1857 on the other. The char pointer returned the correct value, but so did the int pointer with its 1857. How can this be?

There can't be two values for the first byte in the screen buffer! The latter is a true statement. However, the char pointer accesses memory in 1-byte quantities. A char can't represent more than 1 byte of data. On the other hand, the int pointer accesses data in 2-byte increments. The return from an int pointer is not a 1-byte quantity but a 2-byte value. Earlier, it was stated that the monochrome screen buffer used every other byte (even numbers and zero) for character display. The odd bytes are used for character attribute values. In a normal screen, all of the odd bytes have a value of 7. Therefore, the value returned by the int pointer was the 2-byte combination of 65 and 7. The 65, as we already know, is the ASCII code for 'A'. The 7 is the attribute byte. The 2-byte coding for a numeric value of 1857 is, you guessed it, 65 and 7.

In short, the char pointer returned a 1-byte value that was written to the screen as its integer equivalent of 65. This is the only value found in that single byte. The int pointer returned a 2-byte standard int value that was also written to the screen as its 2-byte integer equivalent of 1857. This can also be explained mathematically in the following manner.

```
1857 % 256 = 65    /* First Byte */
1857 / 256 = 7     /* Second Byte */
```

The opposite event takes place when a poke operation is attempted with a pointer of type int. Assuming that *x* is a far pointer with the address of the monochrome screen buffer and of type int, the following expression will cause a value of 65 ('A') to be written to the first byte in the buffer, but a 0 will also be written at the second byte.

```
*x = 65;
```

The first write is what we're looking for. The second definitely is not! The second byte (the odd one) is the attribute byte for the screen display. A value of 0 here causes the character at the first buffer byte to become invisible. The result for a full screen write of all characters 'A' is a blank screen if an int pointer is used for this purpose.

Think of pointers as returning 1 SSU of data or of writing 1 SSU of data. In fact, each accesses 1 SSU in memory. This applies regardless of the type of pointer. However, the SSU for a char type is only 1 byte. Storage unit size for all other data types is 2 or more bytes.

Far pointers specifically introduced into small-memory model programs work just as well for writing or poking data into memory. The following program demonstrates a poke operation using a far pointer.

```
void main(void)
{

      char far *a;

      a = (char far *) 0xb0000000;

      *a = 65;

}
```

This program writes an 'A' to the first byte in the screen buffer and can be compiled by the small-memory model option. It should be stressed at this juncture that the large-model classification of compiler options offers advantages other than being able to address all memory. These include the ability to handle large programs that will not compile using small-memory models. While it is usually preferable to remain within the boundaries of the small-memory model compilers wherever possible, source code size and content will dictate the model that is best suited to a particular program. In some cases, you may have to try both model classifications in order to see which compilation is best for a particular purpose.

If you switch from one to the other, it will be necessary to change the source code slightly when pointer operations that use direct addresses are part of the code. For instance, if you compile a program using the large-memory model, then recompilation with the small-memory model will require that pointers used for far memory access be declared *far* pointers. Going from a small- to a large-memory model is not nearly so tedious. In the large models, a pointer is a 4-byte entity by default. Any pointer that is declared *far* in the large model is unaffected by the modifier, and is still allocated 4 bytes for address storage.

There are also other compiler options that are a combination of both large and small. For instance, the medium memory model in the Borland compiler defaults to far pointers for the code (program) segment, but the data segment defaults to near pointers. This means that the medium model is ideal for compiling C programs with a large amount of source code (larger than can be accommodated by the small-memory model) while still maintaining the efficiency of near pointers in the data segment. No far accesses into memory will be permitted without declaring far pointers for this purpose.

The opposite of the medium model is called the compact memory model. Here, the program segment is limited to near pointers, but all of memory can be accessed via the far pointer default in the data segment. For C programs with only a moderate amount of source code, this model is ideal, especially if many excursions into far memory are anticipated.

Remember our Little Man from Chapter 2. The knapsack of the man who represents a char contains only a single compartment. When a pointer returns an object from a location, the entire contents of the knapsack are emptied. If you figure one item per compartment,

then the little man who represents a char dumps only one item. The int little man dumps two, one from each compartment, the long int little man has four items, etc. A similar thing happens when we try to fill those knapsacks. If you dump one item to a knapsack with four compartments, the remaining three are also filled with zeros, in this case. Those zeros may mean nothing when taken by themselves, but when they are part of an overall code, they mean much more.

Summary

A pointer contains the address of an object in memory. If the address is specified directly as a numeric memory location, then the object at that address can be retrieved. This is a peek operation. If the address is assigned an object value, then the memory location is overwritten. This is a poke operation. Unintentional overwrites can be disastrous, but a poke is intentional and desirable (provided the programmer knows what he or she is doing).

Char pointers are an excellent choice for reading and writing memory locations in a standard fashion. Their 1-byte objects are responsible for this desirability. All other types of pointers may also be used for memory accessing, but one must always remember the SSU size of each type.

Large-memory model compilers automatically assign 4 bytes of address storage for all declared pointers. This allows excursions through all of a computer's memory (assuming typical maximum memory sizes). However, the large model compilers with their automatic far pointers create programs that are usually larger in size than those produced by small-memory model compilers. Also, execution speed with large model compilations is slowed, sometimes considerably.

Most high-quality C compilers for MS-DOS machines now offer the advantage of declaring far pointers from within small-memory model programs. This provides full memory access while still maintaining the advantages of the small-memory model attributes. It should be understood that, even with the small-memory models, 4-byte or far pointers still take longer to execute than do their 2-byte counterparts. However, small model programs that utilize both near (2-byte) and far pointers will execute faster than they would if compiled entirely with a large model, where all pointers would be 4-byte types, regardless of their intended uses.

The old rule still applies. You should have noticed that in all of the program examples, pointers were always initialized before any reading or writing took place. When using far pointers, whether specifically declared from a small-memory model program or from a large model program, the full memory capability can lead to very real dangers if uninitialized pointers are used to write memory. With far pointers, it is now possible to tap in to your system's various interrupts. An unintentional poke into memory can bring about some of the disasters alluded to earlier. Always know where and to what your pointers point.

Chapter 7

POINTERS AND MEMORY ALLOCATION

The first rule of thumb when dealing with declared pointers is to give them something at which to point. This is the pointer *initialization* process, and this rule cannot be repeated too often. All pointers must be assigned a memory address (initialized) before usage, or they will address random memory locations that can literally wreck a program run or possibly even damage disk files.

In most previous program examples, pointers were assigned the memory addresses of other variables. In poke/peek operations, pointers were assigned the addresses of memory locations specifically for the purpose of reading a byte content or writing to a byte of storage.

There are many advantages in using pointers, but in most cases, standard variables are depended upon for safe areas to which the pointers may point. Char arrays are necessary for storing copied strings, but they must be sized to accommodate the largest string length. Wouldn't it be nice if a device were available that would allow memory to be allocated based upon the size of the string to be copied, rather than to some size roughly determined by the programmer and usually oversized for the sake of avoiding any possibility of an accidental

memory overwrite? Fortunately C language offers such conveniences if you know where to look.

Memory Allocation Functions

The two principal memory allocation functions that are a standard part of the C language function library are malloc() and calloc(). They both perform the same basic process of setting aside a safe area of memory to which data may be written, and these data blocks may be of any practical size. Both functions perform operations that check on memory availability, based upon the memory block size requested. When a suitable segment of memory is located, a pointer to the address of the allocated block is returned by the function. The principal difference between malloc() and calloc() is that the latter clears all of the allocated memory, poking zeros into each byte. Malloc() simply allocates the memory without clearing the bytes. The value of each byte is whatever value was already contained there prior to the allocation.

The following program begins our discussion of pointers and memory allocation functions.

```
void main(void)
{

      char a[5][33];

      strcpy(a[0], "Happiness");
      strcpy(a[1], "Data");
      strcpy(a[2], "Now is the time for all good
men");
      strcpy(a[3], "Forget");
      strcpy(a[4], "Computer")

}
```

This program uses a two-dimensional char array sized to store a maximum of five strings, each with a maximum length of 33 bytes, including the NULL terminator. We already know that *a* could have been declared an array of char pointers in order to save on allocated memory. The arrangement above sets aside 165 bytes (5 * 33) for storage to the array.

However, the total storage required for all of these five string constants is only 64 bytes. This means that 101 bytes are wasted in this particular example. The reason for the waste is the manner in which the array declaration must be made. The smallest value allowed in the second part of the subscript must be capable of containing the largest string. However, this large value applies to the other four string storage allocations within the array.

The following program is another example of possible wasted storage space.

```
#include <stdio.h>
void main(void)
{

      int x;
      char a[5][256];

      for (x = 0; x < 5; ++x)
                gets(a[x]);

}
```

This example cannot use a pointer substitute, since no string constants are involved. The gets() function copies bytes retrieved from the keyboard into memory. This program assumes that the longest string will be composed of 255 typed characters and makes room for the NULL with the last. There is no way of knowing just what will be input, so an overly large string length must be assumed for the sake of memory safety.

However, we can conserve memory via the memory allocation functions and the addresses they return to pointers. The following program will allocate storage for all strings input at the keyboard, based upon the length of any string.

```
#include <string.h>
#include <stdlib.h>
void main(void)
{

      char *a, b[256];

      gets(b);

      a = malloc(strlen(b));

      strcpy(a, b)

}
```

The operation of this program is simple. Three char types are declared. Variable *a* is a char pointer, *b* is a char array with a maximum storage capability of 256 characters including the NULL, and malloc() is declared a function that returns a char pointer. The malloc() func-

tion, technically, returns a void. This is a generic pointer type that may be assigned to any other type of pointer without casting. In this program, the void pointer returned from malloc() is directly assigned to the char pointer.

The gets() function uses char array *b* as its storage device. This array can accept up to 255 characters without overwriting memory. The last character is the NULL. Now, malloc() is called, and it uses the length of the string already contained in *b* and returned by the strlen() function to determine how many bytes must be allocated. The strlen() function returns the total number of bytes in the string including the NULL. Therefore, malloc() will set aside just enough storage, no more, no less, to store the string in array *b*. Next, strcpy() is used to copy the string in *b* to the pointer location contained in *a*. This address was returned to *a* by the malloc() function.

Wait a minute! Didn't we just COPY something to a pointer? Isn't this supposed to be taboo?

Ordinarily, the answer is yes. Nothing should be copied to an uninitialized pointer. However, this pointer has the address of a block in memory that has been allocated especially for storage. Therefore, the pointer is initialized. This is a safe area, as malloc() guarantees this. The strcpy() function treats *a* like it would a pointer to a char array. Before each copy, adequate storage is set aside at the address contained in *a*. If there is adequate storage, there is no reason why data cannot be copied into this block of memory. The argument to malloc() is the return from strlen(). This assures that the block is of adequate size. In reality, nothing is being copied to the pointer. Rather, data are copied into the allocated memory block referenced by the pointer.

This program is by no means complete but was simplified to the bare essentials for explanation purposes. The use of memory allocation functions mandates checking their returns for a NULL. These functions return NULL if the amount of memory requested is unavailable. If a NULL return is issued by the function, then the pointer value will be zero (NULL). If this test is not made and a NULL is returned, any attempt to write to the anticipated block of memory (which doesn't exist if the return is NULL) will result in a dangerous memory overwrite. The proper method of handling the operation carried out by the previous code is demonstrated by the following program.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void main(void)
{

      char *a, b[256];

      gets(b);
```

```
        a = malloc(strlen(b));

        if (a == NULL) {
                printf("Out of memory\n");
                exit(0);
        else
                 strcpy(a, b);

}
```

This can be shortened to the code that follows.

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
void main(void)
{

      char *a, b[256];

      gets(b);

      if ((a = malloc(strlen(b))) == NULL) {

            printf("Out of memory\n");
            exit(0);
}
else
            strcpy(a, b);

}
```

In both of these examples, the return value from malloc() is tested for a value of NULL. The actual value of NULL is defined in the stdio.h header file. In most implementations, NULL is assigned a value of 0, but large-memory models require that the value be a long (0L).

When malloc() is called, a conditional test is conducted on the value returned to the pointer. If this value is NULL or 0, then the program displays an "Out of memory" prompt and terminates execution. If the value returned by malloc() is other than 0, then the attempt was successful and this value becomes a pointer to the allocated block.

When large blocks of memory are necessary, the large-memory model compiler versions may be required, as the amount of available memory for allocation is not as limited as

it is with the small models.

Suppose, for some purpose or other, you wanted to create a char array that would hold a very large string, say, 60,000 characters. Don't try a declaration line like the one that follows.

```
char= a[60000];
```

You will quickly find that this exceeds the size your compiler will accept. You may not be able to think of a reason why anyone would require an array of this size, but there are many. For instance, an entire block of characters from a word processor file could be stored in a single variable. The block could then be written or displayed by accessing the same variable. You can't accomplish this through normal declarations, because the space allotted to the stack is just not this large.

The following program shows how this amount of storage could be allocated to one variable unit.

```
void main(void)
{

      char *a;

      if ((a = malloc(60000)) == NULL) {
                  printf("Out of memory");
                  exit();

      }

       /* Remainder of program ..... */
```

You now have a pointer that controls a large block of memory for whatever purpose desired.

The memory allocation functions are certainly not limited to char values. Any pointer of any legal data type may contain the address of a malloc() assignment. The following program demonstrates this principle.

```
#include <stdlib.h>
#include <stdio.h>
void main(void)
{

      int *x, y;

      if ((x = malloc(400)) == NULL) {
                  printf("Out of memory\n");
```

```
                exit(0);
        }
        else
                for (y = 0; y <= 198; ++x)
                        *(x + y) = 88;

}
```

This program declares an int pointer and requires a little explaining. In older C compilers, malloc() usually was fixed to return a pointer of type char. However, more recent versions cause malloc() to return a void pointer, one that is untyped. This means that it can be assigned directly to any type of pointer without casting. A void pointer is generic, a pointer to any type.

The reason for the preference of char pointers for memory allocation lies in the fact that chars utilize 1-byte storage units. Therefore, 400 char units can be stored in a memory block of 400 bytes. However, this program uses an int pointer to access the memory block of 400 characters. Each assignment to *x will result in a 2-byte write. Therefore, only 200 int values can be stored in the allocated memory block. If you try to write 400 integer values, thinking that there is enough allocated memory for this purpose, then you will overwrite the 400 bytes of memory that lie at the end of the block. Very dangerous!

Note that the *for* loop that loads the block (with a value of 88) counts from 0 to 198. Each time an int value is written, 2 of the 400 available bytes in the block are used. It must be understood that the argument to malloc() is expressed in bytes. However, the following construct deals in SSUs.

```
*(x + y)
```

If *y* == 0, then *x accesses the first 2 bytes of the block. However, *(x + 1) does not access the second byte in the block. It accesses the second SSU. Since we are dealing with ints, the SSU is 2 bytes. Therefore, *(x + 1) accesses the second SSU, which falls to the third and forth bytes in the block. The following mini-chart may help in picturing this concept.

```
int   *(x + 0)   *(x + 1)   *(x + 2)   *(x + 3)   *(x + 4)
Byte # 0 , 1       2 , 3       4 , 5       6 , 7       8 , 9
```

This means that the offset value [i.e., 2 in the expression *(x + 2)] can be thought of as half of the block byte position number being accessed. This is not too hard to remember, but different data types (int, long, double, etc.) will also result in different SSUs.

While it has not been stated previously, the standard technique of specifying offsets to pointers, as in *(x + 2), may also be expressed in subscripting brackets. Observe

the assignment below.

```
*(x + 3) = 88;
```

This can also be expressed in the following manner.

```
x[3] = 88;
```

One doesn't often see expressions of this type involving declared pointers, the exception often being pointers that have been assigned the address of a block of memory allocated via malloc() or some other similar function. In programs that use both arrays and pointers, the bracketed offset designations can become confusing. This is the reason for the availability of the more common **(x + 2)* offsetting syntax that is often used with declared pointers. The two methods offer an expressiveness that tends to keep the two types separate in the mind of the programmer. To add to the possible confusion, offsets within arrays can also be expressed using the indirection operator.

```
#include <stdio.h>
void main(void)
{

      int x[40];

      *(x + 0) = 65;

      printf("%c\n", *(x + 0));

}
```

It has been repeatedly stated that arrays and pointers can often be treated as the same in many C operations. These examples highlight the truthfulness of that statement. However, the conventions that have occurred naturally over the years since the C programming language was authored dictate the use of the indirection operator for memory access via pointers and the bracketed subscripts when dealing with declared arrays.

The following program shows a practical usage of malloc() whereby the monochrome screen is read and the contents saved in a memory block. The screen will then be cleared and the contents read back in again. Readers who are using a VGA monitor should change the pointer address to 0xb8000000 for this program to work on their systems.

```
void main(void)
{

      char far *a, *b;
      int x;

      a = (char far *) 0xb0000000;

      if ((b = malloc(4000)) == NULL) {
               printf("Out of memory\n");
               exit(0);
      }

      for (x = 0; x < 24; ++x)
                printf("ANSI C Example of screen
storage.\n");

      for (x = 0; x < 4000; ++x)
               *(b + x) = *(a + x);

      system("cls");
      sleep(5);

      for (x = 0; x < 4000; ++x)
               *(a + x) = *(b + x);

}
```

This program can be compiled using the small-memory model compiler option and is quite impressive even on the slower machines. A char far pointer, *a*, is given the address of the screen buffer. Char pointer *b* is assigned the address of the memory block allocated by malloc(). Note that malloc's return is tested for NULL. The first *for* loop is used to write a sentence to the screen and repeats it so the entire screen is filled. The next *for* loop reads the contents of the screen a byte at a time. As each byte is read from the screen buffer, it is copied to the memory block referenced by *b*. The copy is made on a byte-for-byte basis. This means that the first byte from the screen, **(a + 0)*, is copied to the first byte in the block, **(b + 0)*. This continues until the 4000th byte from the screen is copied to the 4000th block byte. This completes the copy of the entire screen.

At this point in the execution chain, the screen is cleared by calling the MS-DOS CLS routine via the C language system() function. The sleep() function is then called to create a five-second pause. Without this function, the rewrite to the screen occurs so rapidly after

screen clearing that you probably wouldn't be able to see the erasure. The sleep() function lets you plainly see that the screen has been cleared.

After five seconds, another *for* loop is executed, which does the opposite of the previous loop. This time, the contents of the memory block are read back into the screen buffer. The copy takes place on a byte-for-byte basis, and all of the original screen information is rewritten in the blink of an eye.

This program could just as easily have transferred the contents to a disk file, but the process would have taken far longer due to the slow operation of mechanical devices. The reading of the disk information back into the cleared screen would take as long. I have written several commercial software packages for C language tutorial purposes that have used routines similar to the one demonstrated by the last program. These programs contained many tutorial screens. Writing the information to these screens in the usual fashion was too slow and would have created comprehension difficulties for students. The solution was to download all screens to disk files. This was done outside of the tutorial program. When I had all of the screens on disk, I included these files with the tutorial software. During a program run, several screens were loaded into memory blocks. This was done as soon as the program was executed and was part of the overhead time required for loading all information and before the on-screen run started. Then, when a screen was required, it was written directly from a memory block. The write took place almost instantly, and it was possible for the student to switch from one screen to another in a very short time, like flipping the pages of a book.

The following program shows a standard method of saving a screen to a disk file and assumes that the screen buffer already contains the information to be saved.

```
void main(void)
{

      FILE *fp;
      char far *a;
      int x;

      if ((fp = fopen("screen1", "w")) == NULL) {
            printf("Error: Can't open file\n");
            exit(0);

      }

      a = (char far *) 0xb0000000;

      for (x = 0; x < 4000; ++x)
            fputc(*a++, fp);

      fclose(fp);

}
```

This program opens a file in the normal manner, testing for a NULL return, which means that the file could not be opened, possibly due to a full disk, a write protect tab, or some other disk problem. The char far pointer *a* is given the address of the monochrome screen buffer. VGA monitor users will want to change this address assignment to 0xb8000000, the start of the screen buffer. However, this program assumes that, even when using a color monitor, the screen is in text mode (as opposed to graphics mode).

Next, a *for* loop is entered, which steps through each of the 4000-byte offsets in the buffer. While only the even bytes contain the character information, all bytes are read since the entire screen is to be saved. This means that any special text attributes will also be saved. Within the loop, the fputc() function is used to write each byte to the disk file. You will notice that the pointer access is stepped by the increment operator *following* the pointer name. This means that the value in **a* will be returned to the function *prior* to the pointer being stepped to the next position. Here, we are actually changing the address of the pointer. We could also have written this statement in the following manner.

```
fputc(*(a + x), fp);
```

This method adds the value of *x* to the address in the pointer, the latter remaining fixed. Either method is acceptable, but the first increments the pointer address while the second uses the pointer address, unchanged, adding the offset to this value. Both result in the same assignments to the disk file.

When the write is complete, the file is closed, and your disk contains a file named "screen1", loaded with all of the screen information.

You could read this information back into the screen by using the method shown below.

```
void main(void)
{

      FILE *fp;
      char far *a;
      int x;

      if ((fp = fopen("screen1", "r")) == NULL) {
            printf("Error: Cannot open disk\n");
            exit(0);

      }

      a = (char far *) 0xb0000000;

      while ((x = fgetc(fp)) != EOF)
               *a++ = x;
```

```
        fclose(fp);

    }
```

This is a slow method as seen by the viewer's eye, because a character must be retrieved from the file and then written to the screen. This process is repeated 4000 times until the entire screen has been written.

Each value returned to int *x* by fgetc() is assigned as the object at the screen buffer location accessed by *a*. While *x* is an int type that stores all values in 2 bytes of memory and **a* is a char type with only 1 byte of storage, there is no conflict. The value in *x* will be less than 256 and this is a legal assignment to **a*. Since all values of less than 256 are stored solely in the first byte of an int variable (with the second byte being equal to 0) all that occurs is that the second byte value in *x* is simply dropped.

The screen is rewritten, byte-by-byte. This will take some time (depending upon the type of microprocessor and the machine clock speed), and the lags may be very obvious to the screen viewer. The following program shows another method of retrieving this code from the disk. The process takes as long for recovery, but the display write is almost instantaneous.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
void main(void)
{

      FILE *fp;
      int x, y;
      char far *a, *b;

      a = (char far *) 0xb0000000;

      if ((b = malloc(4000)) == NULL) {
               printf("Out of memory\n");
               exit(0);
      }

      if ((fp = fopen("screen1", "r")) == NULL) {
                printf("Error:Can't open file\n");
                exit(0);
      }
```

```
        y = 0;

        while ((x = fgetc(fp)) != EOF)
                    *(b + y++) = x;

        fclose(fp);

        printf("Press any key to write screen");
        x = getch();

        for (x = 0; x < 4000; ++x)
               *a++ = *b++;

}
```

This program may seem to be complex to those new to ANSI C, but it can be broken down into small, easy-to-understand modules.

Two char pointers are declared, far *a* and (near) *b*. The first is given the address of the monochrome screen buffer. The second is handed the address of the start of 4000 bytes of memory cleared by malloc(). At this point, the file containing the screen information written by a previous program is opened. The *while* loop calls fgetc(), which returns the contents of "screen1" in single-byte quantities to variable *x*. Within the *while* loop, the character value in *x* is assigned to the byte accessed by pointer *b*. This byte and all others are within the 4000-byte block set aside by malloc(). Nothing is written to the screen by this action. Rather, the original screen contents stored in this disk file are written to the reserved block of 4000 bytes. This is a relatively slow process, because each byte must be retrieved from the file and then written into memory. This was the same condition found in the previous program, which wrote the screen directly from the disk file.

Once the memory block has been loaded, a prompt appears that tells the user to press any key to write the screen. The getch() function is utilized here. It will halt program execution until a character is received. This character serves no purpose in this program other than to release the getch() execution halt.

When the key is pressed, a *for* loop is entered and the contents from the memory block are written directly to the screen. Pointer *b* returns the byte from the memory block that is written to the screen through access by far pointer *a*. This method was demonstrated in an earlier program.

This program does not get the contents from disk file to screen any faster than did the previous attempt, but the time between actually starting and ending the write is tremendously fast. The time overhead comes from the necessity of loading the information from a disk file.

The thing to do when several pages of information are desired is to load them all in at the same time. Once the screens have been copied to memory blocks, they may then be accessed almost instantaneously. From the user's point of view, it is far more pleasing to initialize and complete the long loading process during the initial call up or execution of the program. Of course, multiple pages will require more memory block allocations. The number of pages that can be loaded into memory will depend upon the amount of memory your machine has and also on the memory model compiler version. Large models are able to allocate more memory than can small models.

Calloc()

Most of the attention given to memory allocation functions has surrounded the use of malloc(). As you will recall, malloc() simply finds a free block in memory and then returns a pointer to it. Calloc() works in a similar fashion, except that it is more convenient when allocating memory for storage units of more than 1 byte. Also, calloc() clears all memory blocks as a part of its operation. This means a block of 4000 bytes allocated by calloc() will return a pointer to a block whose bytes have been set to 0. The following program effectively demonstrates one of the advantages calloc() offers.

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
      int *x, y;

      if ((x = calloc(400, 2)) == NULL) {
            printf("Out of memory\n")
            exit(0);

      }

         for (y = 0; y < 400; ++y)
               *(x + y) = 88;

}
```

An earlier program in this chapter assigned the address of a 400-byte memory block to an int pointer, and int values numbering a maximum of 200 were all that could be written to the block. An int value requires 2 bytes of storage. In the current example, another int pointer is used. Calloc() is called in the following format.

```
calloc(storage_units, unit_size)
```

The arguments express the number of storage units and the unit size. Therefore, the usage of calloc() in this program does not mean that a block of 400 bytes is allocated. Rather, the block is 800 bytes long. The number of storage units is 400, but the size of each storage unit is 2 bytes. Therefore, 400 * 2 = 800 bytes. This program can safely address from 0 to 399 units in this block without overwriting memory. This means that **(x + 399)* will access the 799th and 800th bytes in the allocated block. However, we don't have to worry about this aspect of storage after an allocation for the desired number of storage units has been made. Malloc() requires arguments in the form of bytes. Arguments to calloc() deal with storage units.

Throughout this book, I have been careful to stress that stated storage sizes for various data types apply only to most MS-DOS implementations. This is important because C language offers excellent portability and is often used on mini-computers and very large systems. On these other types of machines, a standard int type may be allocated 4 bytes of storage, a long, 8 bytes, etc. Storage allocations for all data types in C are relative and are based upon the architecture of the machine and many other factors.

The built-in argument protocol of calloc() allows for portability to many other types of machines if the arguments are stated correctly. In the previous program, calloc() was used in the following form.

```
calloc(400, 2)
```

The desire here was to set aside 400 storage units for int values, an int requiring 2 bytes of storage. This is all well and good when dealing with an MS-DOS machine, but suppose the program containing this use of calloc() is to be ported to many other types of computers. There is no portability, because other machines may have different sizes (other than 2 bytes) for int values.

The way to maintain portability is to test the size of storage on the machine that will run the software; that is, base the allocation on the storage size that is tested while the program is running. How is that done?

Simple! Call the sizeof() operator, which has been used sparingly in some of the program examples in this book. This operator returns the size (in bytes) of any variable as well as the size of any data type specified.

```
sizeof(int)
```

This will return the size of an int data type. Here, *int* is a keyword that can be read directly by the function. Now, if this function were executed on an MS-DOS machine, sizeof()

would return a value of 2, but on a mainframe, it might return 4 or more bytes. With this in mind, calloc() could have been used in the previous program in this manner.

```
calloc(400, sizeof(int));
```

Such usage would insure that the correct number of bytes were set aside for the storage of 400 int data types, regardless of the amount of storage allocated to int data types. The calloc() function is already portable. You should make certain that your arguments to it and other such functions are equally portable.

There is nothing magic about the way calloc() performs its task. In fact, it simply calls malloc() as a part of its internal workings, then it assigns each allocated byte a value of 0. The following is an example of how malloc() can be called to exhibit the same portability as calloc().

```
malloc(400 * sizeof(int))
```

Here, the argument to malloc() is a mathematical expression. The number of bytes is expressed as storage units multiplied by the size of the individual unit. This is exactly what calloc() does within its source code structure. The only thing different about these two functions is that when malloc() is used in this manner, it does not reset all allocated bytes to NULL or 0.

As a side note, once an allocated block of memory is no longer needed, it should be freed up for possible allocation to later program allocation requests. This assumes that the program will continue to execute long after the point in the execution chain where the allocated block is no longer needed. To free a block of memory that was allocated with either malloc() or calloc(), use the free() function in the following manner.

```
free(ptr)
```

In this example, *ptr* is the pointer to the block of allocated memory.

Summary

When a pointer is assigned the memory address of an allocated block of memory, we can say that it has been initialized and, in a manner of speaking, has been sized to the number of bytes in the storage area. It is then safe to copy objects to this memory block using the pointer for access. In other words, the pointer may be treated as an array of the same size as the memory block the pointer addresses.

Both malloc() and calloc() will return a NULL value if the amount of memory requested cannot be allocated. Any program that uses either of these functions must test their

returns for the NULL. If you assume that the storage space is available and it turns out that it isn't, then any writes attempted will become memory overwrites with the inherent dangers that they present.

The ANSI C standard requires that malloc() and calloc() return pointers of type void. This means that the return address value can be assigned to any legal pointer data type. Casting is unnecessary.

Through the use of C language pointers and the memory allocation functions, it is convenient to perform block reads and writes. These were demonstrated in this chapter by programs that read, wrote and rewrote the monitor screen. Such operations are certainly not limited to the visual display screens and may be used to read and write any block of memory anywhere in the system. The reason for the screen buffer examples is that the memory writes can actually be seen. This is far more expressive, far more visual from a tutorial standpoint.

Chapter 8

POINTERS AND FUNCTIONS

While pointers are indispensable throughout all phases of ANSI C programming, their capabilities yield high-powered program performance, especially when dealing with programmed functions. We already know that arguments are passed to and from functions by value. It is not possible to change the value of a calling program variable within a function unless we have the address of a variable in the calling program.

ANSI C functions have the capability of returning values that can be, in turn, assigned to variables within the calling program. However, through pointer operations, we can write specialized functions that will return a value to the calling program and change the values in other variables whose memory addresses are handed to the function. In this manner, it is possible to write a function that, figuratively, returns many values in a single operation.

Again, the direct programming of pointers is what gives C language its power, expressiveness, and authority. It is also the pointer that has vaulted ANSI C to the top of the list of languages most desirable for software development.

The following program incorporates a special function to return the square of a value passed to it. This is a useless function for any but tutorial purposes, but as the discussion continues, other examples will provide some practical application tools.

```
#include <stdio.h>
int square(int i);
void main(void)
{

      int x, y;

      x = 8;
      y = 11;

      x = square(x);
      y = square(y);

      printf("%d  %d\n", x, y);


}
int square(int a)
{


      return(a * a);

}
```

The square() function is passed a value from the calling program. This value is assigned to *a*, a variable that is internal to the function. The function return value is *a* * *a*, or the square of the passed value. This program will display the following values.

```
64  121
```

Under main(), the square() function is called twice because it is necessary to obtain the square of two different values. It is not possible for a C function to directly return two values. In this example, the function "equates" to the square of its passed value.

This next example shows how a similar function can accept the value of a calling variable and the memory address of another. The passed value is returned by the function, just as in the example above. However, the address is used to reassign the value in another variable to its square. The result is a function that returns a square value in the normal manner and rewrites one via a pointer.

```
#include <stdio.h>
int square(int i, *ii);
void main(void)
{


      int x, y;

      x = 8;
      y = 11;

      x = square(x, &y);

      printf("%d  %d\n", x, y);

}
int square(int a, int *b)
{


      *b = *b * *b; /* rewrite memory */
      return(a * a);/* return square */

}
```

This program will also display 64 and 121 on the screen, but the function was called only once, which results in a faster program run. In the function, *a* is an auto variable of type int, but *b* is a pointer of type int. Notice that the memory address of *y* in the calling program was passed to the function. The value of this address was assigned to pointer *b* in the function. The following expression will look a bit weird at first glance due to the number of asterisks.

```
*b = *b * *b;
```

The first two asterisks are indirection operators accessing the object at the address passed to the pointer. The third asterisk is the multiplicative operator, while the fourth is another indirection operator. If *b* were a standard (auto) variable, then this line would have been written in the following manner.

```
b = b * b;
```

The use of a single symbol to name operators that perform differently in C is a very good reason to stick to the source code formatting conventions that were established many years ago. This means that all arithmetic operators should be separated from their variable names by a space.

```
x = a * b;
```

This expression is far easier to decipher than the one that follows.

```
x=a*b;
```

The former provides much better clarity of understanding. The multiplication of pointer *b* could also be written in C shorthand.

```
*b *= *b;
```

This means the same as the following expression.

```
*b = *b * *b;
```

Either way, there are still a lot of asterisks. But, suppose these examples did not adhere to the source code formatting conventions. We might end up with the following glob of source code.

```
*b=*b**b;
```

Alternately, consider the previous shorthand method, now bastardized by poor program formatting.

```
*b*=*b;
```

If you have not been adhering to the formatting standards displayed in the original Kernighan and Ritchie, *The C Programming Language*, then perhaps you might wish to familiarize yourself with these rules of etiquette for the C programmer. I have observed that, as ANSI C grows in popularity, there is a breakdown in these formatting conventions. Even many of the commercial references that accompany C compilers are getting sloppy in this regard. This is not elitism on my part but an appeal. The writing of sloppy source code, regardless of the fact that it compiles and runs properly, can quickly become a detriment to newcomers learning this language. Also, it's getting harder for those of us who are supposed to know something about C to figure out some of the garbled programs we see displayed. More on this later!

The example of a function returning a value, supposedly for assignment to another

variable in the calling program, and changing the value of another calling program variable is not especially unique, although it does not follow a typical function programming pattern. More than likely, the operation would be handled by sending two address values to the function and returning void. The following program demonstrates this.

```
#include <stdio.h>
void square(int *i, int *j);
void main(void)
{

      int x, y;

      x = 8;
      y = 11;

      square(&x, &y);

      printf("%d  %d\n", x, y);

      }
void square(int *a, int *b)
{

      *a *= *a;
      *b *= *b;

}
```

The same result is had within the calling program, and the function source code is a little more typical of what one might expect.

In all of these examples, the function expected a specific number of arguments and had to be passed this same number in order to prevent an error, but what about functions that accept a varying number of arguments? A good example is printf(), which can be used in seemingly different formats including those that follow.

```
printf("hello, world\n");
printf("%s\n", "hello, world");
printf("%d\n", x);
printf("%d  %c\n", x, y);
```

How is this apparent bevy of arguments handled? The following program begins a discussion that will answer part of this question.

```
#include <stdio.h>
void newprint(char *c, int i);
void main(void)
{

      int x, y;

      x = 17;
      y = 134;

      newprint("%d  %d", x, y);

}
void newprint(char *str, int args)
{

      printf("%s\n%d", str, args);

}
```

This program declares and assigns two int variables and then calls a new function, arbitrarily named newprint(). The arguments to newprint() are handled in the same manner that would apply if a similar call to printf() were involved. The newprint() function source code at the bottom of the calling program declares only two arguments in its parentheses, *str* and *args*. The first is declared a char pointer, the second is an int. This is a test program/function purely for discussion purposes, so the first job at hand is to see what the two arguments yield. We'll use the standard printf() function to display the values of the two arguments. The screen will display the following.

```
%d  %d
17
```

This means that *str* points to the string that was enclosed in quotation marks within the original call to newprint(). We learned much earlier that a quoted constant in C actually returns the address of that constant in memory. Therefore, *str* points to the string constant "%d %d". We can say, loosely, that *str* is that quoted string from the calling program. Variable *args* is displayed as 17, the value of the first variable argument to newprint(). However, you will recall that two variable arguments were provided. Only one is displayed. The reason for this is that our printf() function within newprint() called for only one int argument in its format control string. However, *args* is only a single argument, so what do we do?

Here is where pointers become invaluable. Though this program doesn't show it, the address assigned to *args* references both variable arguments originally made to newprint. For now, we will say that the address of *args* references a block of memory that contains all of the values handed to newprint() during the initial call. The first 2 bytes contain the value in *x* (17) from the calling program. The second 2 bytes contain the value in *y* or 134. These values are aligned sequentially at the start of storage reserved for *args*. Incidentally, this storage was set aside automatically by the calling program when the newprint() function was invoked. Without going into laborious detail, let's just say that this is the manner in which C handles function calls.

The following program reveals more about this discussion.

```
#include <stdio.h>
void newprint(char *, int);
void main(void)
{

      int x, y;

      x = 17;
      y = 134;

      newprint("%d  %d", x, y);

}
newprint(char *str, int args)
{

      int *ptr;

       ptr = &args;

      printf("%s\n%d\n%d\n", str, *ptr, *(ptr + 1));

}
```

This program will display the following on the monitor.

```
%d  %d
17
134
```

This is what we are looking for. The big change in the function source code lies in the use of *ptr*, a pointer of type int that has been declared within the function. It was stated earlier that

the address of int variable *args* references all of the second argument group passed to newprint(). The first argument group is the formatting string enclosed in quotes. Therefore, the storage address of *args* is the place to begin. Pointer *ptr* is given the address of *args* via the address-of operator. Now, **ptr* contains the object value of the first argument to newprint() (after the format string), and *(*ptr* + 1) contains the second.

Let's shuffle the deck a bit by changing the calling arguments slightly in the program that follows.

```
#include <stdio.h>
#include <string.h>
void newprint(char *, int);
void main(void)
{

      int x, y;
      char a[20];

      x = 17;
      y = 134;

      strcpy(a, "Today");

      newprint("%d  %d  %s", x, y, a);

}
void newprint(char *str, int args)
{
      int *ptr;

      ptr = &args;

      printf("%s\n%d\n%d\n%s", str, *ptr, *(ptr +
1), *(ptr + 2));

}
```

This produces the following monitor display.

```
%d  %d  %s
17
134
Today
```

Again, **ptr* returns the 2 bytes that equate to decimal 17; **(ptr + 1)* gives us 134, and **(ptr + 3)* returns the address of the storage location of Today. In short, the storage address starting at the address of variable *args* forms a memory chain of actual values and/or addresses of strings.

Of course, we are cheating a bit, as the supposed purpose of newprint() is to replace the printf() function found in all versions of C. Since newprint() calls printf(), this new function doesn't really take its place. The purpose of this discussion is to illustrate argument alignment in function calls, but to avoid being accused of "copping out," I will explain the workings of printf() in a little more detail.

The format string in printf() is the controlling element for reading the second argument grouping content. First of all, remember that if we want to print the letter 'C' on the screen, this letter does not reside, as a letter or character, anywhere in memory. It is stored, as are all data, as a numeric value. The ASCII value of 'C' is 67 decimal.

Printf() source code is long and complex because the function has to read the contents of the format control string, access memory for the type of data specified, and then build up a character string accordingly. If the first element in the control string is a %d, the printf() source code instructs the function to access **ptr* (using the above example as a reference) or **(ptr + 0)*, if you prefer, and retrieve an int value. If the contents of the first 2 bytes addressed by the pointer are 17 and 0, this means that the numeric value equal to 17 is to be displayed as a decimal integer. However, we are writing to the screen, so we can't simply poke in a value of 17 at a character location. Instead, printf() has to place the character '1' followed by the character '7' in the display string it is building internally. The single number 17 is represented as two characters, a '1' and a '7'. The '1' is represented by ASCII 49 and the '7' by ASCII 55. Therefore, the values that must be poked into the display screen buffer to represent numeric 17 are 49 and 55. This is a long conversion from 17 and 0, the 2-byte storage code for the number 17. This process applies to all other numeric values as well, with the larger numbers requiring a larger number of characters to represent them.

After the first two characters representing numeric 17 are placed at the front of the display string being built by the function, the format control string is read again, starting at a point just following the %d. Assume the next read yields a space. This signals no special memory access code, so it is simply displayed as the whitespace character that is printed when the keyboard space bar is pressed. The space character has a value of ASCII 32, so this value is added to the screen display string that now appears as follows.

```
49 55 32
```

The next format string read yields another %d. This signals printf() to go to the next pointer position, **(ptr + 1)*, and retrieve 2 bytes of integer data. The value returned goes through the same conversion process previously discussed. We'll assume the retrieved integer value is 28, which yields an ASCII character value of 50 and 56. The screen string building in printf()

now reads as follows.

```
49 55 32 50 56
```

Again printf() goes back to the remainder of its format string. Let's assume that it next discovers another whitespace character. This character signals no special code, as does %d, for example, so the character is to be displayed as a space on the screen. The ASCII code for a space is 32. See how the character string has grown.

```
49 55 32 50 56 32
```

Once again, the format string is read, and this time a %s pops up. This signals a string value and printf() goes to the next argument vector of **(x + 2)*. Here, it looks for an address of a string, a pointer. It accesses that address location from the 2 bytes used by the pointer for storage (the assumption here is that a small-memory model compiler defaulting to 2-byte pointers is used) and copies the numeric byte content to the end of the screen string it is building. If we assume the string content is ABC, then 65, 66, and 67 are copied to the internal screen display string. The string has grown considerably.

```
49 55 32 50 56 32 65 66 67
```

Assuming that this is the end of the format control string, printf() is now ready to dump these contents. The following program should be familiar as it pokes bytes to the monochrome screen buffer.

```
void main(void)
{

      char far *a, s[10];
      int x, y;

      a = (char far *) 0xb0000000;

      s[0] = 49;
      s[1] = 55;
      s[2] = 32;
      s[3] = 50;
      s[4] = 56;
      s[5] = 32;
      s[6] = 65;
      s[7] = 66;
```

```
        s[8] = 67;

        y = 0;

/* Access every other byte in buffer */
            for (x = 0; x < 18; x += 2)
                    *(a + x) = s[y++];

}
```

This program will display the following string on the monitor.

```
17 28 ABC
```

This has been a laborious process and doesn't even begin to reveal all of the complex workings of printf(). Please understand that this explanation has been a very brief and technically symbolic representation of the processes that a function like printf() must go through in order to deliver the screen output that we take for granted. This discussion has also revealed how a function can be written to handle an indeterminate amount of passed argument values. Without pointers, such operations would not be available to the programmer.

The concepts explained lead to other possibilities, such as functions that will perform mathematical operations on any number of arguments. The add() function is demonstrated by the following program.

```
#include <stdio.h>
int add(int i);
void main(void)
{

        int a, b, c, d, e, f;

        a = 12;
        b = 4;
        c = 8;
        d = 234

        e = add(4, a, b, c, d);
        f = add(3, b, c, d);

        printf("%d\n", e);
        printf("%d\n", f);

}
```

```
int add(int args)
{

      int tot, x, y, *ptr;

      ptr = &args;
      y = *ptr++;
      tot = 0;

      for (x = 0; x < y; ++x)
      tot += *ptr++

      return(tot);

}
```

This program assigns values to four int variables and then calls add() two times in order to return the total of all its arguments to variables *e* and *f*. This program displays the following numeric values on the monitor.

```
258
246
```

The add() function accepts any number of integer arguments, but the first argument *must* state the number of arguments passed. The first call to add() uses four values to be added, so the first argument is 4. The second call uses a first argument of 3 in order to add the three values in the variable arguments that follow.

There is no format string in this function. None is needed, as the first argument is all that is necessary to effect the add operation. The only argument shown as being passed to add() is named *args* and resides within the body of the function. As before, *args* is the start of memory where all arguments to add() have been placed. The declared pointer is given the address of *args* and the first value is extracted. The following expression does two things.

```
y = *ptr++;
```

First, the object value at the address contained within the pointer is returned to *y*, then the pointer address is incremented by one SSU, in this case, 2 bytes. The value in *y* is the first argument to add(). On the first call to this function, *y* is equal to 4. Variable *y*, then, contains the total number of arguments to be added by the function. Another internal variable, *tot*, is assigned an initial value of 0. This variable will contain the added totals.

Next, a *for* loop is entered that increments *x* from a value of 0 to a value that is one less

than *y*. Remember, *y* contains the number of arguments to be added. Counting from 0 to *y - 1* will cause the loop to cycle the number of times in *y*. The loop count begins at 0, not one. Assuming 4 arguments to be added, looping from 0 to 3 *(y - 1)* will yield four loop cycles.

Pointer, *ptr*, has already been incremented to the next data position before the loop is entered. This incrementing was handled during the object assignment to *y*. On the first pass of the loop, **(ptr + 1)* is accessed for a return value of 12 (the value of the passed argument in variable *a* from the calling program). The pointer is actually advanced by the increment operator, but **(ptr + 1)* is another way of expressing its relative position. This value is added to the current value of *tot*, which is 0 at this stage. On the next pass, **(ptr + 2)* is accessed and added to the current value of *tot*. This variable is now equal to 16 (12 + 4). On the next pass, the 8 is added and on the next, 234 is added to the total in *tot*. The loop times out because it has cycled four times. The value in *tot* is now equal to the sum of the four arguments. This value is returned to the calling program, where it is assigned to *e*.

The next call to add() contains only three values to be added, so the first argument is 3. The same operations begin all over, except that the loop will cycle only three times due to the new value in *y*. This time, the total of the three arguments to add() will be returned.

Pointers to Functions

A C language function is not a variable, but you can declare pointers to functions that can then be used in a manner in which other pointers are used, such as placing them in arrays, passing them to other functions, etc. One doesn't see a lot of pointers to functions, but this technique can sometimes be used to advantage. One common usage of pointers to functions is in purposely hiding the intent of a program. A pointer to a function serves to hide the name and source code of that function. Muddying the waters is not normally a purposeful routine in C programming, but with the security placed on software these days, there is an element of misdirection that seems to be growing.

The following program demonstrates the basic techniques of declaring a pointer to a function.

```
#include <stdio.h>
#include <string.h>
void main(void)
{

      int x, scarp();
      char a[15];

      strcpy(a, "COMPUTER");

      x = scarp(a, strlen);
```

```
        printf("%d\n", x);

}
int scarp(char *c, int (*girl) (char *c))
{

        return((*girl) (c));

}
```

This process may look a little ridiculous, and maybe it is, but it does effectively demonstrate a pointer to a function. In this case, the function in question is strlen(), which returns the length of a string argument as an int.

Within the calling program, another function is invoked. Scarp(), arbitrarily named, passes two values. The first is the char array that now contains "COMPUTER". The second is the name of the strlen() function. In this usage, *strlen* without parentheses is the address of the strlen() function.

Within scarp(), the address of *a* is assigned to char **c*, and the address of *strlen* is passed to a pointer to this function that is arbitrarily named *girl*. Note the declaration of this entity within the function.

```
int (*girl) ()
```

This says that *girl* is a pointer to a function that returns an int value. At this juncture, the expression *(*girl)* may be used exactly as you would strlen().

```
return((*girl) (c));
```

The expression above means exactly the same thing as the one below.

```
return(strlen(c));
```

**(girl)* () points to the storage address set aside for strlen() when it is declared within the program.

Functions Returning Pointers

For some reason, there is a kind of fear among beginning ANSI C programmers in regard to pointers in general. I hope the discussions presented here have lessened these concerns. However, the subject of functions that return pointers seems to be even more frightening. There is no reason for such a fear.

It has been shown previously how pointer arguments to functions can be used to alter a value in the calling program. In these cases, no pointer values were returned from the function. The address value passed to the function was used for accessing a variable's address to make changes in memory.

The following program calls a function that returns a pointer of type char.

```
#include <stdio.h>
#include <string.h>
char *combine(char *, char *);
void main(void)
{

      char a[10], b[10], *p;

      strcpy(a, "horse");
      strcpy(b, "fly");

      p = combine(a, b);

      printf("%s\n", p);

}
char *combine(char *s, char *t)
{

      int x, y;
      char r[100];

      strcpy(r, s);
      y = strlen(r);

      for (x = y; *t != '\0'; ++x)
      r[x] = *t++;

      r[x] = '\0';

      return(r);

}
```

This program will display the following string on the monitor.

```
horsefly
```

The combine() function is nothing more than a version of strcat() that returns a pointer to the combined string and doesn't alter the value of any of its argument objects.

The program declares two char arrays, a char pointer, and *combine() to be a function that returns a char pointer value. This declaration must be made to let main() know what type of return is expected. Two string constants, "horse" and "fly", are copied to the two arrays. Next, combine() is called in the following format.

```
p = combine(a, b);
```

Now, *p* is a pointer of type char. Therefore, it must be handed the address of a char object. This means that the return from combine() must be a char pointer.

Skipping to the function source code, we see that the function is declared for a char * return by the opening line.

```
char *combine(s, t);
```

Upon return, this function will equate to a char pointer. Variables *s* and *t* are declared char pointers by the next line in the function.

Within the function body, *x* and *y* are declared ints, and *r* is declared a char array with 100 bytes for storage. The subscript value need only be adequate to store the strings of *s* and *t* combined.

The strcpy() function is called to copy the string pointed to by *s* into *r[]*. Next, strlen() returns the number of characters in *r[]* and assigns this returned int value to *y*. A *for* loop is entered that assigns *x* a starting value of *y*. Since *y* is equal to the number of bytes in *r*, and *r* presently holds the contents of the string pointed to by *s*, this means that *x* will start its count at a value that is equal to the position of the NULL byte in *r[]*. This occurs because *x* is assigned an initial loop value of *y*, the latter being equal to 5.

To put it another way, the string pointed to by argument *s* has been copied into the *r* array. The end of this string is terminated by a NULL. Strlen() returns the number of characters in *r[]*. We know that *s* points to "horse", a string containing 5 characters. When "horse" is copied into *r[]*, strlen(r) returns a value of 5. The character in *r[]* at the fifth offset *(r[5])* is the NULL.

When the loop starts cycling, the initial value in *x* is 5. On the first pass, the NULL byte in *r[]* is overwritten by the first byte referenced by *t*, as evidenced below.

```
r[x] = *t++;
```

Pointer *t* is incremented by 1, and the loop cycles again. On the next pass, *x* is equal to 6, so byte *i[6]* receives the next character in *t.

On each pass of the loop, the exit clause is tested.

```
*t != '\0'
```

Many programmers think that the second expression in a *for* loop must set the maximum value (or minimum value in negative-going loops) for the loop variable, in this case, *x*. This is not true! This portion of the loop statement is an exit clause and may contain any legal expression. In this example, the loop will terminate when **t* is equal to NULL or '\0'. Each time *r[x] = *t++* is executed within the loop, pointer *t* is incremented. Before another loop cycle, the expression **t* != '\0' is evaluated. When the end of the string referenced by *t* is reached, the loop will be exited.

Since we know that *t* points to "fly", we also know that the loop will cycle three times. After the third pass, the exit clause brings about a loop exit. However, the loop variable has already been incremented in readiness for the next pass. Even though this pass never came about, *x* is set to the next logical byte in *r[]*.

Since the termination of the loop occurred when the NULL character referenced by *t* was encountered, this NULL was never written to the end of *r[]*. Therefore, the contents of *r[]* encompass a series of discrete characters and not a string. A NULL terminator is required to make a string, so the contents of *r[]* are made into an official C string by the following assignment.

```
r[x] = '\0';
```

This puts the finishing touch on the whole operation. Array *r[]* now stores a bona fide string.

The rest is quite simple. All we have to do is pass a pointer to *r* back to the calling program.

```
return(r);
```

This expression is the obvious and correct choice. Remember, *r* used without subscripting brackets is a pointer to the contents of this array. This address value is passed back to the calling program and assigned to pointer *p*. When *p* is handed to the printf() function, the display of "horsefly" is the end result.

The complexities with this function lie in manually concatenating "fly" to "horse". The return of a pointer by the function is child's play. The following is a commented listing of the above function that may clarify a bit more of its operation.

```
/* Return a char pointer to calling program */
char *combine(char *s, char *t)
{

     int x, y;    /* x and y are ints */
```

```
    char r[100]; /* char array with 100 bytes */

    strcpy(r, s); /* copy s into r */
    y = strlen(r);  /* return length of string */

    /* count x starting at y = x */
    for (x = y; *t != '\0'; ++x)
    r[x] = *t++; /*read byte from t into r*/

    r[x] = '\0';    /* tack on NULL */

    return(r);     /* return a pointer */

}
```

Again, this program calls a function that returns a char pointer; it does not alter the object values of either of its two pointer arguments.

Summary

C language functions may be thought of as completely separate programs. When called by any other entity, the function takes control. This control is not returned to the calling entity until the function returns a value (via the return statement) or until the function execution chain is completed.

Arguments are passed to functions *by value*. If *x* is an argument to a function, it is the value that *x* contains and not the variable and its assigned storage area that is passed. A like variable is created within the function to store the value in *x* that was passed.

We do not have access (within the function) to the variables themselves when they are used as arguments to functions. However, the address of a variable or other object in memory may be passed to a function, again by value. When the function knows the address of an object in memory, then it can directly address that object and make changes or retrieve information. If the value of an object is to be changed by a function, then the address of the object must be passed and not the value of the object.

When multiple arguments are passed to a function, an argument list is contained in memory. Within the function, a pointer to the first argument can be used to access all of the list, provided that there is a specific indication of what kind of and how many arguments were passed to the function. The inner workings of the printf() function access arguments in this manner, with the format string specifying the exact nature of the arguments that follow.

Pointers to functions are possible and sometimes practical. While a pointer is not a variable, it is possible to define pointers to functions that can be passed to other functions, placed in arrays, and used to manipulate data.

Functions are often used to read and write data in the calling program by being passed the memory address of an object. Likewise, it is quite simple to write a function that actually returns a memory address or pointer. To do this, the function itself must be declared in the calling program as a type that returns a pointer, as is shown below.

```
char *combine();
```

In addition, the source code of the function must contain a heading of the return type.

```
char *combine(char *s, char *t)
```

These declarations inform the calling environment of the return intentions of the function.

The ability to incorporate new functions easily is a strong aspect of the ANSI C language. Since these functions are actually separate programs that have no relationship with the calling program, other than through the arguments that are specifically passed back and forth, pointers provide a direct means of communication. Through the act of passing memory addresses, the function is provided a pathway into certain areas of the calling program. The function can then make changes and exchange information. Pointers are quite often the true power behind C language function operations.

Chapter 9

Pointers and Structs

The pointer types discussed so far in this book have all been of standard data types and usually were directed to point to the address of a declared variable or to a known memory location to be read or written.

Of course, a pointer can also be given the address of a function and point to it. If you think about it long enough, you will soon realize that anything within a program lies in memory, therefore, a pointer can point to any part of the program, any device driver, any device interrupt.

Structs in the C programming language are a derived data type comprised of a collection of intrinsic data types. It is the specific collection of these intrinsic types that actually forms the derived data type. While ANSI C is not an object-oriented language, the struct approaches the definition of a true object more so than does any other element of this language. Another ANSI C entity, the array, also provides a means of combining several variable storage areas into one accessed unit. In a struct, however, data do not have to be of the same type, as is the case with an array.

The following code illustrates the most common form of an ANSI C struct.

```
struct dbase {
      char client[30];
      int acctnum;
      double amount
};
```

This is a template for a structure named *dbase*. The structure is a collection of variables of different data types. The first struct element is named *client* and is a char array. The second and third are numeric types named *acctnum* and *amount*. This collection of variables may be accessed via the structure tag name, *dbase*. The following program shows one method of doing this.

```
#include <stdio.h>
#include <string.h>
void main(void)
{

      struct dbase r;

      strcpy(r.client, "Jones, M.);
      r.acctnum = 12;
      r.amount = 23917.33;

      printf("%s\n%d\n%ld\n",r.client,r.acctnum,
r.amount);

}
```

The following expression defines a variable named *r*, which is of type dbase.

```
struct dbase r;
```

Within the body of the program, the structure member operator "." connects the structure name (*r*) and the member name. Therefore, *r.client* accesses the structure element *client[]* and is used as the target argument to strcpy(). The int and double members are assigned values, using the same operator.

To display the contents of each structure element, printf() is invoked. Again the structure member operator comes into play with each of the arguments to this function.

Pointers to structures are very common in the C programming language, so common that a special operator is used for accessing structure elements by a pointer. The pointer to a struct member operator -> is used to access a member of a structure using a pointer variable.

The following program is a copy of the one above, except the structure elements are accessed via a pointer.

```
#include <stdio.h>
#include <string.h>
void main(void)
{

      struct dbase r, *q;

      q = &r;

      strcpy(q->client, "Jones, M.");

      q->acctnum = 12;
      q->amount = 23917.33;

      printf("%s\n%d\n%ld\n", q->client, q->acct-
num, q->amount);

}
```

In this program example, the declaration below names *r* a variable of type dbase and *q* a pointer of the same type.

```
struct dbase r, *q;
```

The next program line assigns *q* the address of *r*. Pointer *q* now points to the storage address reserved for struct dbase.

The pointer to a struct member operator is used to access each structure element. This special operator is simply a convenient shorthand method of access when dealing with pointers.

```
q->client
```

This expression means exactly the same thing as the one below.

```
(*q).client
```

Here, the structure member operator in conjunction with the indirection operator is used for access. The parentheses are necessary because the structure member operator "." takes

higher precedence than does the indirection operator.

Pointers to structures are necessary when passing a whole structure to a function. ANSI C passes arguments to functions by value, so it is necessary to pass the address of the structure to the function to gain access to the data elements. The following program shows how this may be accomplished.

```
struct test {
      int a;
      char b;
      char name[20];
};
#include <stdio.h>
#include <string.h>
void load(struct test *);
void main(void)
{

      struct test r;

      load(&r);

      printf("%d  %c  %s\n", r.a, r.b, r.name);

}
void load(struct test *s)
{

      s->a = 14;
      s->b = 'A';

      strcpy(s->name, "Group");

}
```

A structure named *test* is defined with three elements of type int, char, and char array. Variable *r* is declared to be of type struct *test*. The load() function is called and is passed the memory address of *r* using the address-of operator. Within the function, *s* is declared a pointer of type struct *test*. This pointer has the address of the struct and uses the pointer to struct member operator to access each of the three struct members. Since *s* is a pointer, assignments are made to the memory locations of each of the elements. These values are written directly to the structure. Upon returning to the calling program, the values *14 A* and *Group* will be written to the screen.

When dealing with ANSI C structs, there is no problem with individual access of each element. However, when the structure as a whole is to be passed as an argument, a pointer to the structure is required. Pointers to structures have their own connective operators that allow for access in a convenient, shorthand manner. Trying to access a structure via a function call that does not include the structure address is an error and will result in garbage writes to and reads from the structure.

Comparing Structs and Arrays

Earlier in this chapter, a comparison was made between ANSI C structs and arrays. An array is an entity designed to store multiple quantities, all of which are of the same data type. For instance, a char array is designed to store a continuous series of char data types. If the array is sized to contain 10 chars and the memory address of the beginning storage location is at 100, then the first element is stored at this location, the second is stored at 101, the third at 102, etc. The data are stored sequentially.

The same sequence applies to an array of ints, except that 2 bytes of storage are allocated for each element. Therefore, an int array whose initial storage address is 100 will store the first int value in bytes 100, 101. The second int value will be stored at 102, 103, the third at 104, 105, etc.

An array of floats with the same beginning address will store the first value in bytes 100, 101, 102, 103. The second quantity will be stored at 104, 105, 106, 107. The same 4-byte storage applies to long int types.

In every case, the storage units are aligned sequentially. An entire block of continuous memory is allocated for storage to any array, regardless of the types of data the array is to contain. The actual memory allocated to the array is dependent upon two factors: the data type and the value in the array subscript.

For char arrays, 10 chars may be stored in 10 bytes. For int arrays, 20 bytes will be required to store 10 int values. With doubles, a total of 80 bytes will be required, as a double-precision floating-point value is stored in 8 bytes of memory. All of these quantities assume a standard MS-DOS implementation.

The key element in all of this is that allocated storage for arrays is sequential, one continuous block of memory. We do not run into a situation where the first element in an array is stored at an address of 100, while the second is stored at 1500. If the latter storage routine were the case, accessing individual elements in an array would be very difficult, as the address of each element would have to be obtained by separate operations. In ANSI C, we know that if ten elements are a part of a char array, each of these elements can be accessed as a sequential offset of the first element.

From a storage point of view, ANSI C structs are closely akin to arrays, in that all of the members are stored sequentially. Consider the following struct template.

```
struct employee {
      char class;
      int tenure;
      double salary;

};
```

This derived type contains three intrinsic data types, a char, an int, and a double. If we assume that the address of the first array member (*class*) is at location 100, then storage for int *tenure* begins at 101, offset 1 byte from the beginning storage location. This offset is produced, because 1 byte of storage is required for the char data type.

Since an int consumes 2 bytes, storage for double *salary* begins at address 103, offset 2 bytes from the int member. Bytes 103 through 110 are allocated for double *salary*, since a double value requires 8 bytes of storage.

The important thing to remember is that the storage for an ANSI C struct is sequential. Therefore, a struct is closely akin to an array in this regard. The only difference lies in the fact that different data types may be stored in a struct. We might think of a struct as an array of multiple quantities that may be of different data types.

The struct begins with a template. This is an appropriate name because the template is used to allocate memory. Such allocation takes place by reading the number of members and their individual data types. The number of one data type is multiplied by the individual storage requirements for this data type to arrive at the necessary storage. When the storage values for the total number of each data type in the struct are added, the overall storage requirement for one instance of the struct is obtained.

Now that the program has this information, it can properly allocate memory for as many struct instances (variables) as are declared in the program. For instance, struct *employee* requires 11 bytes of total storage. Therefore, if a program declares four variables of type *employee*, a grand total of 44 bytes will be allocated in four discrete groups of 11 bytes each. We can say that such a program would have four instances of struct *employee*.

The following program details this allocation of structs.

```
struct e_record {
      char name[15];
      int id;
      int years;
      double sal;

};
#include <stdio.h>
#include <string.h>
void main(void)
```

```
{

      struct e_record emp;

      strcpy(emp.name, "Ken Jones");

      emp.id = 14;
      emp.years = 22;
      emp.sal = 535.00;

      printf("Employee Name: %s\n", emp.name);
      printf("Employee Number: %d\n", emp.id);
      printf("Years Employed: %d\n", emp.years);
      printf("Employee Salary: $%-.2lf\n", emp.sal);

}
```

When this program is executed, the following information will be displayed on the monitor.

```
Employee Name: Ken Jones
Employee Number: 14
Years Employed: 22
Employee Salary: $535.00
```

This program directly accesses struct members via the struct variable and the struct member operator. The struct pointer version of this program is shown below.

```
struct e_record {
      char name[15];
      int id;
      int years;
      double sal;

};
#include <stdio.h>
#include <string.h>
void main(void)
{

      struct e_record e, *emp;

      emp = &e;
```

```
        strcpy(emp->name, "Ken Jones");

        emp->id = 14;
        emp->years = 22;
        emp->sal = 535.00;

        printf("Employee Name: %s\n", emp->name);
        printf("Employee Number: %d\n", emp->id);
        printf("Years Employed: %d\n", emp->years);
        printf("Employee Salary: $%-.2lf\n", emp-
>sal);

    }
```

In this code, a struct variable is created along with a pointer of type struct *e_record*. Next, the pointer is given the address of the struct variable (*e*). The struct variable is necessary because the compiler will allocate space when it is declared. Without the struct variable, the struct pointer has no size (is not initialized). Remember, a struct template simply tells the compiler how much space to allocate for the block of memory that will serve as storage for any struct variables. No automatic allocation occurs until a struct variable is declared.

We can write the previous program in a slightly different manner by using the malloc() function to allocate the needed storage space. The program below does this, thus eliminating the need for a struct variable. The struct pointer is initialized via a call to malloc().

```
struct e_record {
        char name[15];
        int id;
        int years;
        double sal;

};
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void main(void)
{

        struct e_record *emp;

        if ((emp = malloc(sizeof(struct e_record)))
== NULL)
                exit(0);
```

```
            strcpy(emp->name, "Ken Jones");

            emp->id = 14;
            emp->years = 22;
            emp->sal = 535.00;

            printf("Employee Name: %s\n", emp->name);
            printf("Employee Number: %d\n", emp->id);
            printf("Years Employed: %d\n", emp->years);
            printf("Employee Salary: $%-.2lf\n", emp-
>sal);

    }
```

This program will execute properly and will display the same screen data as the previous two examples. Instead of creating a struct variable and then handing its address to the struct pointer, only the pointer is declared. Next, malloc() is called to allocate the necessary space for the pointer. Note that the return from malloc() is tested for the occurrence of a NULL value. This would mean that memory could not be allocated, so the program would terminate due to the exit() function being called.

The following expression is a simplified version of what is actually seen in the program source code.

```
emp = malloc(sizeof(struct e_record));
```

The malloc() function is called with an argument that uses the sizeof() operator with struct *e_record*. This tells malloc() to allocate the space needed to contain the struct. The sizeof() operator returns the total struct size just as it can return the size of an int, a long, a double, etc. simply by using the type name. The total size of the struct is garnered from the struct template. In this program struct *e_record* is a derived type. This type has size just as does any other data type, derived or intrinsic.

Storage is automatically allocated for a struct when a struct variable is declared. However, if a struct pointer is declared, there is no automatic allocation, as is the case with any other type of declared pointer. Allocation for a struct pointer requires the use of a memory allocation function such as malloc(). Without the call to malloc() in the program above, any assignments would be made to an uninitialized pointer, a dangerous condition.

Even though the sizeof() operator can be conveniently called to return the size of the struct, let's calculate the storage requirements for ourselves. Examine the struct template below.

```
struct e_record {
      char name[15];
      int id;
      int years;
      double sal;

}
```

From the earlier discussions in this chapter, you should now be able to determine the exact size of struct *e_record*.

The calculations proceed as follows.

```
char name[15] =        15-bytes
int id =               2-bytes
int years =            2-bytes
double sal =           8-bytes

Total Size =           27-bytes
```

The size of struct *e_record* is 27 bytes. The char array with 15 elements requires 15 bytes. Each int type requires 2 bytes, and the double type will require 8 bytes. Therefore, a variable of type *e_record* will (or must) be allocated 27 bytes of storage. This is the size of this derived data type.

We can verify this by placing the following statement in the previous program.

```
printf("%d\n", sizeof(struct e_record));
```

This will write a value of 27 to the monitor screen, the size that we have already calculated for struct *e_record* storage.

Now, let's compare this struct with an array. We know that the allocated storage for the struct is consecutive. Therefore, the first 15 bytes (of 27 total bytes) will be reserved for the char array member. The next 2 bytes are reserved for the first int member, while the 2 bytes that follow are reserved for the second int member. The last 8 bytes are allocated for the double member.

The following program will prove the contention that struct data are stored in a single, consecutive block of memory.

```
struct e_record {
      char name[15];
      int id;
      int years;
```

```
        double sal;

};
#include <stdio.h>
#include <string.h>
void main(void)
{

        struct e_record emp;

        strcpy(emp.name, "Ken Jones");

        emp.id = 14;
        emp.years = 22;
        emp.sal = 535.00;

        printf("%u\n", &emp.name);
        printf("%u\n", &emp.id);
        printf("%u\n", &emp.years);
        printf("%u\n", &emp.sal);

}
```

On the system used for testing all applications in this book, the screen displayed the following.

```
65498
65513
65515
65517
```

Some simple math will show that the memory addresses are offset by the size of the data declared in the struct template. The first address in the listing above is the start of the *name* member. The next address is at a 15-byte offset from this address, this being the size of the char array. This second address is the location of the *id* data member, an int, so there is a 2-byte offset from this location to the next storage address for the *years* member, which is also an int. The last address is offset 2 bytes from the *years* location. We can safely assume that the end of the memory block allocated for the struct is located at address 65525, since 8 bytes are required for storage of a double value.

The program that shows the address location of each member in the struct has simply used printf() with pointers to each struct member accessed by the struct variable, *emp*. Instead of returning the object value at each location, the address-of operator (&) causes each of the struct variable designations to return its address. This proves that ANSI C structs

store their data in sequential blocks of memory.

The following program is a slight modification of an earlier one. It writes object values directly to the memory locations allocated for the struct. Following these assignments, the struct member object values are accessed by using the structure member operator.

```
struct e_record {
      char name[15];
      int id;
      int years;
      double sal;

};
#include <stdio.h>
#include <string.h>
void main(void)
{

     struct e_record emp;

     char *c;
     int *i, *h;
      double *d;

      c = &emp;  /* Assign address in emp to c */

      i = c + 15; /* offset for id */
      h = c + 17; /* offset for years */
      d = c + 19; /* offset for sal */

/* Copy string into offset 0 of struct */
      strcpy(c, "Ken Jones");

/* Assign object values to offsets */
      *i = 14;
       *h = 22;
      *d = 535.00;

/* Extract struct data in normal manner */
      printf("Employee Name: %s\n", emp.name);
      printf("Employee Number: %d\n", emp.id);
      printf("Years Employed: %d\n", emp.years);
      printf("Employee Salary: $%-.21f\n", emp.sal);

}
```

This program will display just what the previous examples did, but notice that the struct was never loaded with data in the traditional manner. Rather than accessing the allocated memory locations via the structure member operator (.) to make these assignments, these operations have taken place using separately declared pointers.

Initially, the address of the struct variable named *emp* is assigned to *c*, a pointer of type char. Next, the address of pointer *c* is used as the base reference for address assignments to two pointers of type int and one of type double. It is necessary to align the pointer types with the types of data they are to store. A char pointer was chosen for the base reference because of the offsets. Char pointers operate in units of 1 byte. If *c* had been declared an int pointer, then an offset value of *c + 15* would have resulted in an actual offset of 30 bytes from the base location. This occurs because pointers are incremented in SSUs, not (necessarily) in bytes. Therefore, incrementing a 2-byte int pointer by 15 SSUs would result in an actual 30-byte offset. The convenience of the 1-byte nature of char types is utilized here, although a void pointer could be made to serve as well, since it is also a 1-byte entity.

Once the proper offset addresses have been assigned, the memory locations referenced are assigned object values. Since *i* and *h* are int pointers, the object values will be encoded into 2 bytes. This is what we want. The double pointer, *d*, will store its object value in an 8-byte encoded format. Copying the string to the char pointer is handled in a normal fashion using strcpy().

At this point in the program, the struct members have been fully assigned. What has actually happened in this program is that pointers have been used to invade the allocated memory set aside for the exclusive use of the struct variable. Even though separately declared pointers were used to access the memory locations allocated to the various struct members, the end result is exactly the same as if the assignments had been made using the conventional structure member operator in conjunction with the struct variable and the member names.

To prove this, the next portion of the program is identical to the earlier example from which it was derived. Each printf() function is handed its argument using the structure member operator and the struct member names. The end result is the familiar display.

Hopefully, this roundabout way of dealing with structs will alert the reader to some of the many intricacies that occur internally in the program whenever structs are accessed. You can be sure that internal pointer operations are in great abundance when structs are made a part of an ANSI C program. Fortunately, the process is invisible, and most programmers don't really appreciate all that goes on behind the scenes when structs are programmed. However, delving into the minuscule operations of almost any programming element that is a part of ANSI C will eventually reveal the hidden pointer operations that make the processes appear outwardly simple.

Summary

In ANSI C, a struct is a collection of intrinsic data types that result in a single, derived data type. In many ways, a struct may be compared with an array, since storage is allocated for all

struct members in one block of sequential memory. Storage for each member is based upon its individual requirements. Member storage within the allocated block is arranged in the order in which such members are declared within the struct template.

However, arrays are limited to storing data that are all of the same type. This limitation is not placed on a struct which stores a collection of different data types as a single unit.

Structs are such specialized tools that special operators have been developed to access struct data. The struct member operator (.) directly accesses struct members, while the pointer to a struct member operator (->) accesses struct members via pointers. Both means of access have their own special features and will often be utilized by ANSI C programmers.

Pointers to structures are absolutely essential when passing the struct to a function. Arguments to functions in ANSI C are passed by value, so the address of the struct must be handed to any function that is expected to effect operations on any of its members.

Chapter 10

Pointers and Unions

Many of the same principles already discussed in Chapter 9 apply equally to ANSI C unions. A union template is identical to a struct template save for the fact that the *union* keyword is used. Union members are specified in the same fashion. We even use the same operators for accessing union members.

Both a struct and a union allow for the combining of different data types in order to create a derived type. However, a union does not have the capability of simultaneously storing more than one member value, whereas a struct may store all member values at a time.

A union is a simple device that allows data of any type to be stored at a single memory location. When a union is created, an area of memory is allocated for storage. All union members reside at the same address. It is for this reason that any assignment to a union member automatically overwrites any previous assignment made to any other member of this same union.

Using a union, different data types may be grouped as a collection, but only a single value may be retained by the union at any one time. Consider the following union template.

```
union alpha {
      int i;
      char a;

};
```

This union has two members, an int and a char. The total storage for a union is handled in a totally different manner than is storage allocation for a struct. With the latter, total storage is based upon the number of members of each data type multiplied by the storage requirement for each type. Storage allocation for a union is far simpler. Total union memory allocation is based upon the minimum memory required to store its largest (from a storage viewpoint) data type.

In the above example, the union will be allocated a total storage quantity of 2 bytes, because the largest data member is an int, requiring 2 bytes in most MS-DOS configurations. This quantity is adequate to store an int value and more than adequate to store a char value, the latter requiring only 1 byte.

Consider the following program.

```
union alpha {
      int i;
      char a;

};
#include <stdio.h>
void main(void)
{

      union alpha r;

      r.i = 128;
      r.a = 'A';

      printf("%d\n", r.i);

}
```

This program will display a value of 65 on the monitor screen and not 128. This occurs because an assignment to any union member automatically overwrites any assignment made to any other member within this same union. In the example above, the int member is assigned a value of 128. For the sake of discussion, let's assume that this value is stored in the 2 bytes beginning at address 100. Next, the char member is assigned a value of 'A'.

When the printf() function is called to display the value in the int member, this value has changed from the initial 128 to 65. The reason for this is that 'A' has an ASCII value of 65. This value overwrites the single byte at address 100, which was originally used to store 128. Byte 101 stored a 0, 128 and 0 being the 2-byte (int) code for 128. When printf() accesses bytes 100 and 101, the 2-byte encoding is now 65 and 0, the int encoding for a value of 65 decimal.

This proves that an ANSI C union is allocated one starting address for all union members and that only one member may contain a useful value at any one time.

In simplest terms, a union contains a set of members that share the same memory address. The union members are accessed via the struct/union member operator (.) or a struct/union pointer to a member operator (->).

The following program example performs some simple operations using a union.

```
union test {
      int x;
      float y;
      double z;
};

void main(void)
{

      union test a;

      a.x = 19;
      a.y = 131.334;
      a.z = 9.9879;

}
```

Nothing is actually displayed on the monitor when this program is executed, as it is totally involved in assigning variables.

We might think of a union as a generic pointer to a specific memory location that may be used to store all types of data at this single address. In this program variables *x*, *y*, and *z* share the same memory address. Each union member assignment overwrites the one made previously. The last union member assignment is the only one that remains in effect. The storage space needed to store the largest union member (double) is 8 bytes.

If the ANSI C union sounds more and more like a specialized pointer, that's because it is. As a matter of fact, the program above can be completely rewritten using pointers. The end result will be the same. The pointer version of this program is shown below.

```
#include <stdlib.h>
void main(void)
{

      int *x;
      float *y;
      double *z;

      if ((z = malloc(sizeof(double))) == NULL)
            exit(0);

      y = (float *) z;
      x = (int *) z;

      /* x, y, and z all contain the same address */
      /* memory allocation is for a double type */

      *x = 19;
      *y = 131.33
      *z = 9.9879;

}
```

This works in the same manner as the previous example that used the union. Here, three pointers of type int, float, and double are declared. The malloc() function is called to allocate enough memory for a double data type. Next, each of the pointers is handed the address in *z*. At this juncture, they all point to the same memory location, an area that has been allocated enough storage to contain a double-precision floating-point value. Like a union, the variables all share the same address, and memory has been allocated for the largest variable type. Also like a union, each time an object value is assigned, it overwrites the previous object value of any other variable. In a union, only one member is active at a time, and this condition has been set up in the last program by using pointers, all of which have been handed the same address.

While this works as well as the union example, this is true only from an execution standpoint. Obviously, using a union is far more efficient, because all of the storage and addressing is handled by the union declaration. The access allowed by the union avoids the use of convoluted pointer operations such as those shown above.

The following program declares a union pointer and accesses data members using the pointer operator.

```
union test {
      int x;
      float y;
```

```
        double z;
};
#include <stdio.h>
#include <stdlib.h>
void main(void)
{

        union test *ptr;

        if ((ptr = malloc(sizeof(union test))) == NULL)
                exit(0);

        printf("%d\n", sizeof(union test));

        ptr->x = 19;
        ptr->y = 131.334;
        ptr->z = 9.9879;

}
```

This program will display a value of 8 on the monitor, because this is the size of the storage allocated to this union. As we already know, the maximum size of any union is the same as the size of its largest data type, in this case, a double-precision floating-point type.

Unions and Memory Conservation

The simple program examples used for tutorial purposes in this book do not place a strain on memory, at least in normal execution environments. Certainly, a premium can be placed on memory conservation when program code begins to take on gigantic proportions. However, what do we mean by "gigantic proportions"?

This is a relative term that is best referenced against the amount of memory available within a system. Some commercial programs go to great (and expensive) lengths to conserve memory use during execution. One factor is the desire to stay within the 640K base memory limit to avoid having to use extended or expanded memory. When these other areas of memory must be utilized, one must consider the practical limit of machines that typical buyers may be using. As of this writing, a 2-megabyte limit is placed on may types of high-level general applications software. This means that all programs written for this class of buyer should require no more than 2 megabytes of RAM to be fully operational. For other buyer classes, other limits may be (somewhat arbitrarily) assumed. For instance, high-level CAD (computer-aided design) programs typically require a minimum of 4 megabytes of RAM, with 8 and even 16 megabytes being required for many of the latest offerings in this area.

Another factor that can enter the picture that relegates even the smaller programs into

the "gigantic" category. These are called *TSRs* for terminate-and-stay-resident and *ISRs* for interrupt service routine. Such programs remain in memory at all times, consuming RAM while other programs are executed in the foreground. When triggered (often by the touch of a single or multiple key combination), these memory-resident programs become active.

Owing to the fact that such programs constantly consume a portion of RAM, it is essential that they be made as small as possible. This is a situation where every byte of savings is precious. Given such a set of circumstances, the ANSI C union becomes a very valuable programming tool.

To repeat, a union is allocated one block of memory, sized to the requirements of its largest single data member. The largest single intrinsic data type is a double. However, unions and structs may also contain arrays of any data type as a data member. Here, size can grow to monstrous proportions. Structs can also be members of other structs or of unions. Union members may even consist of other unions. The capability of combining many different data types, intrinsic and derived, allows both structs and unions to form very complex (and often, large) derived data types.

The following program does not use a union. Rather, it depends upon two structs to amass data types.

```
struct e_record {
      char name[15];
      int id;
      int years;
      double sal;
};

struct a_record {
      char name[15];
      char address[40];
};


#include <stdio.h>
#include <string.h>
void main(void)
{

      struct e_record v;
      struct a_record w;

      strcpy(v.name, "Bill Collins");

      v.id = 44;
```

```
        v.years = 12;
        v.sal = 351.22;

        printf("Name: %s\n", v.name);
        printf("Identification: %d\n", v.id);
        printf("Tenure: %d years\n", v.years);
        printf("Salary: $%-.2lf\n\n", v.sal);

        strcpy(w.address, "523 Short St.; Front
Royal, VA  22630");
        strcpy(w.name, "Bill Collins");

        printf("Name: %s\n", w.name);
        printf("Address: %s\n", w.address);


}
```

This example is similar to several others that were discussed in the last chapter. The *e_record* struct is the same as before. However, the *a_record* struct has been added, and it contains two data members, a char array sized to 15 bytes to store a name and another char array of 40 characters to store an address. This is a tutorial example that has been made as simple as possible. The *a_record* struct might just as easily have included 20 or more data members.

In this usage, the structs are incorporated to help in grouping the needed data for some type of employee record. We will assume that the employee's address is kept separate from the other data for some reason such as creating a mailing list.

When this program is executed, it will display the following on the monitor.

```
Name: Bill Collins
Identification: 44
Tenure: 12 years
Salary: $351.22

Name: Bill Collins
Address: 523 Short St.; Front Royal, VA  22630
```

This program utilizes two structs and declares a variable for each struct type. How much storage has been allocated for the structs?

We already know from the last chapter that struct *e_record* will be allocated 27 bytes. Calculating storage for struct *a_record* is simple, as we simply add the two subscript values in the array declarations, arriving at an allocation figure of 55 bytes. Total storage for both structs is 82 bytes.

Notice that the second struct is utilized only after the first struct has been loaded and accessed by the printf() function. At this point in the program, struct *e_record* is no longer needed. However, it has been created, and its memory allocation will remain in effect until the program is terminated.

Now, let's bring a union into the picture to try to improve on memory consumption.

```
struct e_record {
      char name[15];
      int id;
      int years;
      double sal;
};

struct a_record {
      char name[15];
      char address[40];
};

union com {
      struct e_record a;
      struct a_record b;
};

#include <stdio.h>
#include <string.h>
void main(void)
{

      union com v;

      strcpy(v.a.name, "Bill Collins");

      v.a.id = 44;
           v.a.years = 12;
      v.a.sal = 351.22;

      printf("Name: %s\n", v.a.name);
      printf("Identification: %d\n", v.a.id);
      printf("Tenure: %d years\n", v.a.years);
      printf("Salary: $%-.2lf\n\n", v.a.sal);

      strcpy(v.b.name, "Bill Collins");
```

```
        strcpy(v.b.address, "523 Short St.; Front
Royal, VA  22630");

        printf("Name: %s\n", v.b.name);
        printf("Address: %s\n", v.b.address);


}
```

The display from this program is exactly like that created by the one that preceded it. However, what is our memory consumption now that the union has been made a part of this latest example? Examine the union template below.

```
union com {
      struct e_record a;
      struct a_record b;
}
```

This union contains two data members, each a struct. We know that *e_record* is allocated 27 bytes of storage and that *a_record* is allocated 55 bytes. However, this is a union, so storage is allocated for only the largest data member. Therefore, the total storage allocation is 55 bytes, not 82 bytes as was the case when two structs were utilized without the union.

This means that there has been a savings of 27 bytes, because only a single union variable is declared. This represents an approximate 33 percent decrease in storage for the struct elements in comparison with the earlier program. This is a sizable percentage factor. However, it won't be appreciated in a simple program of this size. When data groupings grow to 100K or more bytes, this process becomes far more valuable.

Unions allow one area of memory to be used for the storage of many different program data elements. Obviously, multiple use of the same memory location is going to result in a savings when compared with storing all data elements in their own exclusive memory areas.

If we were to include the following line in this program, the size of the union would also be displayed during the execution run.

```
printf("%d\n", sizeof(union com));
```

The display value would be 55, the storage requirement of the largest struct in union *a_record*.

Summary

Unions operate in a manner that can be compared to a special pointer that may be used to reference a memory location for the purpose of storing any type of data, intrinsic or derived.

While a struct template and a union template are declared in much the same manner, there are vast differences in how each of these stores data.

A union is allocated a memory block that is adequate to store its largest data member. Any assignments to any member automatically overwrite any previous assignments to any other union member. This occurs because all assignments to union members are stored at the same location.

Unions may be incorporated for many purposes, especially where it is necessary to control the size of memory allocations. Of course the manipulation of such data must lend itself to the limited duration of data existence, since every assignment made to a union member is overwritten by any other assignment that may follow.

Due to the specialized nature of a union and to its parallel operation with ANSI C structs, the two entities share the same access operators. What was described earlier as the structure member operator (.) becomes the union member operator when used to access union members. The same applies to the struct/union pointer operator.

Chapter 11

Pointers to Pointers

Pointers have addresses too. I'm not referring to the addresses they are assigned as objects but to the exclusive address of each pointer that is declared. An earlier discussion on memory addressing stated that small compiler models used 2-byte pointers, while large models used 4-byte pointers. This refers to the exclusive area set aside for each declared pointer to store an address object assigned to it. We know that all standard variables have two values. The first is an address value, or the place in memory where data are stored. The second is an object value, or the data that are stored at the exclusive address.

What makes pointers a little more difficult to understand is that their object values are memory addresses. However, like all variables, pointers have two values as well. The first is the address value where storage is set aside for the pointer to store its object. The second value is the object value, but in this case, the object is another address.

It's obvious that pointers must have a fixed address to store their objects. How else could data be stored except in memory? To further illustrate this point, observe the following program.

```
#include <stdio.h>
void main(void)
{
      int *x;
      char *a;
      long *y;
      double *d;

      printf("%u  %u  %u  %u\n", &x, &a, &y, &d);

}
```

Here's our old friend, the address-of operator (&), which is used in front of variable names to return their address values. In this example, the same operator is used in front of pointer names to return their address values. Therefore, as unusual as it may sound, the following expression is a pointer to a pointer.

```
&x
```

This names the memory address that pointer *x* uses to store its object value, which is another memory address. In most programming assignments, it is seldom necessary to access the exclusive address used by pointers, but the following program shows what is possible. This example exercises the pointer-to-a-pointer concept using a rather roundabout method.

```
#include <stdio.h>
#include <string.h>
void main(void)
{

      char *b, c[10];
      int x, *a;

      strcpy(c, "Goodbye");

      b = "HELLO";
      a = &b;
      x = &c[0];
      *a = x;

      printf("%s\n", b);

}
```

During execution, what will this program display on the screen when *b* is handed to printf()? Variable *b* is a char pointer that is assigned the address of the constant "HELLO". The correct answer is the following string.

```
Goodbye
```

This is displayed when the object at the address in *b* is written to the screen by printf(). How can this be?

First of all, we must look at strcpy(), which copies "Goodbye" to *c*, a char array. Next, *b* is assigned the address of "HELLO", the constant value that you probably thought would be displayed by this program. However, the plot thickens!

Pointer *a* is now assigned the address of pointer *b*, using the address-of operator to gain access to the exclusive storage area of the latter pointer. The expression *&b* is a pointer to a pointer. Now, *x*, an auto variable of type int, is assigned the address of the char array. No, *x* is not a pointer, but there's no law that says it can't be assigned a memory address, as long as that address is in the normal integer range. In this usage, *x* doesn't point to *c*, because a standard variable can't point. The value stored in *x* also happens to be the same value as an address location, but this is not a pointer to that address. In other words, you cannot directly access array *c* via *x*, but you can access it indirectly.

The final stage is writing the address value that is contained in *x* to the memory location pointed to by *a*. Remember, *a* has the address of the exclusive storage area for pointer *b*. This area has been invaded, violated, and changed to a new address, this being the address of array *c* that points to the bytes containing "Goodbye". Notice that *a* was declared a pointer of type int. This means that **a = x* writes the value in *x* as a 2-byte integer to the address pointed to by *a*. Two bytes are assigned to pointers using the small-memory model compiler option, so an int pointer is the logical type to use for changing the object address in any pointer.

All of this is "going around the barn" to simply assign *b* the address location of *c*. The easy way to effect this operation would be to replace all of the code following

```
b = "HELLO";
```

with

```
b = c;
```

That's all the varied assignments and hidden manipulations did in the program above.

In fact, the ridiculous avenues traveled in this latest example are on a par with the contents of a char array being changed by a function that has been passed the address of the array. The only difference here is that the memory address OBJECT of *b* was being altered

by a pointer that had *b*'s memory address. To aid you further in understanding the previous program, it is presented again with comments.

```
#include <stdio.h>
#include <string.h>
void main(void)
{

      char *b, c[40];
      int x, *a;

      strcpy(c, "Goodbye"); /* Copy constant into c */

      b = "HELLO";  /* assign b memory address of HELLO */

      a = &b;     /* read STORAGE address of b into a */
                 /* Note: a is an int pointer; b is a char
pointer */

      x = &c[0];  /* Variable x is assigned address VALUE
*/
                /* x is not a pointer. Its object value */
                  /* is an integer that equals the address
*/
                /* of &c[0]. */

      *a = x; /* Write value in x as an int (2-bytes) to *a
*/
              /* *a accesses two bytes of memory as its
storage unit */

      printf("%s\n", b);  /* Display what b points to */

}
```

Declared Pointers to Pointers

The concept of a pointer having the exclusive storage address of another pointer is, perhaps, a bit exotic, but this is nothing to get alarmed about in C programming. In most applications, you don't see pointers to pointers very often, but they can and do exist.

The former program was an example of doing this within the framework of what had already been previously discussed. However, a pointer to a pointer is a valid variable in ANSI C and is specially declared. The following program demonstrates this.

```
#include <stdio.h>
void main(void)
{

      int x, *p, **ptp;

      x = 454;
      p = &x;
      ptp = &p;

      printf("%u  %d\n", *p, **p);

}
```

In this program ***ptp* is declared a pointer to a pointer of type int. This means that *ptp* expects to be handed the address of a pointer. Variable *x* is assigned a value of 454, then its address is assigned to pointer *p*. Next, the address of pointer *p* is assigned to pointer to a pointer *ptp*. The printf() function is invoked to better illustrate the results. Think of the pointer to a pointer in the following manner.

```
*(*ptp)
```

This may make the concept a little easier to comprehend. In any event, this program will display the memory address of pointer *p* and then the object value of 454. When dealing with pointers to pointers, we must keep track of the indirection operators. The following expression accesses the object of the pointer whose address is held by the pointer to a pointer.

```
**ptp
```

However, the expression below returns the address assigned to the pointer to a pointer.

```
*ptp
```

We can even make the situation more bizarre and say that the expression below is a *pointer to a pointer to a pointer.*

```
&ptp
```

The following program demonstrates the use of pointers to pointers of type char.

```
#include <stdio.h>
#include <string.h>
void main(void)
{

      char a[20], *b, **c;

      strcpy(a, "Capacitor");

      b = a;
      c = b;

      printf(%s  %c\n", *c, **c);

}
```

This program will display the following string.

```
Capacitor  C
```

Remember, a pointer to a pointer is a pointer within a pointer as well. The indirection operator is used to differentiate the objects being sought. If we think of *c* in this program as **(*c)*, then **c* returns the address handed to *c* in the following assignment line.

```
c = b;
```

***c* accesses the ultimate object that the "pointer which is pointed to points to." That's quite a mouthful, but what we are faced with here is a system of stacked pointers. Most programmers refer to these as *nested pointers*, but thinking of them as stacked usually provides better comprehension. It is the object that is nested and *not* the pointers. At the bottom of the heap in a pointer stack is the ultimate object. Above that object is a pointer that points to it. Above that pointer is a pointer to a pointer. To get to the object, you have to start at the top with the pointer to a pointer, go through the pointer, and finally access the object.

In the program above, **c* is the memory address of the nested object, passed to the pointer to a pointer by the pointer whose address it was assigned. The expression ***c* is the object at the bottom of the stack. In order to display a string, the string address is handed to printf(). This is what **c* is in the above program, while ***c* is the 1-byte object at the same address that returns the first character in the string.

In ANSI C, we can create as high a stack of pointers as we desire. The following program reaches utterly ridiculous proportions.

```
#include <stdio.h>
void main(void)
{

      int x, *p, **ptp, ***ptptp, ****ptptptp;

      x = 274;
      p = &x;
      ptp = &p;
      ptptp = &ptp;
      ptptptp = &ptptp;

      printf("%d\n", ****ptptptp);

}
```

As you might have guessed, this program displays the object value of 274 on the screen.

Understand that this is not a practical program. The chances of encountering such a monstrosity in a working program are very, very slim. However, this example does demonstrate the deeply nested capabilities of pointer operations, and it's prudent to understand the principle of pointers to pointers in the event that an application arises that can make use of the concept.

Summary

Like all variables, pointers have two values. The one most discussed is the right value or object value. However, a pointer's object value is an address. We normally think of the lvalue as being an address. A pointer also has an lvalue, which is the address location of storage assigned exclusively to a pointer in order for it to store its object that is the address of (presumably) another memory location.

When a pointer is declared to receive the lvalue or address of another pointer, then it is known as a pointer to a pointer. We can stack these pointers many layers high, although most applications do not take the nesting to the extent some examples did in this chapter.

While you probably will not find a pressing need to resort to pointers to pointers, it is important to understand the concept so that this feature of the ANSI C language may be utilized when best programming efficiency and expression demand such operations.

Chapter 12

Source Code Format

This final chapter may go unread, unnoticed, and unappreciated, but it contains information that can make a major difference in understanding the ANSI C language and all of its many exotic, interesting, and powerful aspects. The most basic level of understanding any language begins with the examination of source code. How this source code is presented seems to be taken for granted by a growing number of ANSI C programmers, which can create some very difficult problems.

When C language was first introduced, its source code was presented in a simple, easy-to-read format. The following program is an example.

```
#include <stdio.h>
#include <string.h>
void main(void)
{

      int x;
      char a[40];

      strcpy(a, "Format");

      for (x = 0; x <= 10; ++x)
      printf("%s\n", a);

}
```

This program is very easy to comprehend, because it is written in a style that expresses its intent. The main() function falls on the left hand margin. The opening brace is on this same margin immediately below the function name. Then, there is a blank line, a separation.

Next, the declaration block appears, indented five spaces. All declarations are made in this block, which is then differentiated by another blank line.

The next block is also indented five spaces and is used for immediate assignments to variables. In this example, strcpy() is used to write a copy of the constant into array *a*. Another blank line serves to separate the initial assignment block from the next program operation.

A *for* loop is presented in the normal fashion. Its single object of control is a printf() function. Note that this line is indented five more spaces. The closing brace signaling termination of the execution run under control of main() is again on the left margin and separated by a blank line. The opening and closing braces encompassing the code under control of main() are easily identified because of their placement in relationship to other program statements.

While the expressiveness of C means many things, the presentation of the source code is a major part of this, to my way of thinking. Now, let's take the previous program and write it in a most inappropriate manner.

```
#include <stdio.h>
#include <string.h>
void main(void){
int x;
char a[40];
strcpy(a,"Format");
for(x=0;x<=10;++x)
printf("%s\n",a);}
```

Is this program easy to understand? My immediate answer is NO! It will compile and run exactly as did the previous program. The compiler sees no difference at all. As a matter of fact, this entire program could have been written on a continuous line with no difference in operation. However, the compiler sees a program in one way and we humans see it in another. The example above has no order, it has no expressiveness, and it is possibly an overexaggeration of the way many people are writing ANSI C programs today.

I came to C programming from a BASIC language environment. One of the major problems with BASIC interpreters of the older genre lay in the impossibility of easily deciphering its source code, especially when code size ran to several hundred lines. There was little order.

A major influence on the learning of the C programming language is the ability of the student to comprehend small portions of a complex program. At first, such a program may

be extremely simple from the experienced programmer's point of view. However, the student is "working in the dark" in many areas that are almost instinctive to the seasoned programmer. The only solution is to have the student break down each program into blocks. If these blocks are not clearly delineated, his or her job is increased tenfold or more.

Proper C language source code format calls for a space between numeric operators and their left and right values, which has been demonstrated throughout this entire book. Let's examine the following program as an example.

```
#include <stdio.h>
void main(void)
{

      int h, i, *x, *y, z;

      i = 10;
      z = 20;

      x = &i;
      y = &z;

      h = *x * *y / ((float) *y / *x);

      printf("%d\n", h);

}
```

This program is arranged according to good formatting principles. The assignment block has been broken down into three parts. The first assigns auto variables, the second assigns memory addresses to pointers and the third assigns *h* the value of several mathematical operations on the previous variables. The block could have been written in one section without the blank line separators, but to me, this makes the program less understandable.

Now, let's take this same program and reformat it in the manner some programmers tend to use.

```
#include <stdio.h>
void main(void)
{

      int h,i,*x,*y,z;
      i=10;
      z=20;
```

```
        x=&i;
        y=&z;
        h=*x**y/((float)*y/*x);
        printf("%d\n",h);

}
```

What we have here is a hodgepodge. The program is so simple that it can be deciphered with a bit of study if you know C language fairly well. If not, it may take longer. The point is that the original program from which this poorly formatted copy was made is immediately comprehended. The operation of the previous program literally jumps out at anyone reading the source code who has even a minimal C background.

The whitespace between each character in the assignment and declaration lines cleans up the source code. The following line of source code is almost frightening!

```
h = *x**y/((float)*y/*x);
```

How do we differentiate the multiplicative operator from the indirection operator? They both use the same symbol (*). Or is *y* in this example a pointer to a pointer using two indirection operators? This program will execute in a normal fashion, but the formatting of the source code is garbage.

GIGO (garbage in, garbage out) is a term coined to describe computer output based upon erroneous input. The same applies to a human being. This type of notation is confusing. Maybe the student simply fails to comprehend anything about the program and gives up in disgust. Even worse, he or she may garner an incorrect impression of what is taking place and proceed onward armed with this faulty assumption. This leads to multiplied difficulties and is one of the prime reasons why some programmers simply drop C language completely, averring that it is a ridiculous language with no form whatsoever.

Consider the following example, which is even worse.

```
#include <stdio.h>
void main(void)
{
int h,i,*j,x,*y,**z;
x=14;
y=&x;
z=&y;
i=22;
j=&i;
h=*j***z/**z;
printf("%d\n",h);
}
```

The assignment to *h* is a killer. What does it mean? If it had been written in the following manner, perhaps this question would be unnecessary.

```
h = *j * **z / **z;
```

This still wouldn't be the easiest expression to decipher, but it would be far simpler.

In order to learn ANSI C, you must write your source code in a manner that allows for visual comprehension. A source code format has been established to provide the necessary clarity. It was not arrived at arbitrarily. I admit that when I first took up C language, having come from a BASIC interpreter environment, I did most of the awful things alluded to in previous examples. I also got nowhere on my first try. I was one of those programmers who gave up in disgust after my initial efforts met with failure. The second time around, I decided to do it like they did it in K&R's *The C Programming Language.* This was the most beneficial thing I ever did for myself in regard to learning C, although I wasn't aware of it at the time.

When I devote an hour or so to teaching proper formatting while conducting a C language seminar, I get all kinds of resistance. "If it doesn't make any difference to the compiler as to how many whitespaces and blank lines there are in a program, why go to all the extra bother?" is what I often hear, especially from BASIC programmers who usually hug the left margin with all of their program lines. Or, "As long as I understand what I write, why use a format developed by someone else?"

These questions are legitimate, and they are asked out of ignorance, not laziness (usually). The answer to both of these is that you have to be able to understand your own code, and you will not if you don't stick to an organized, symmetrical, expressive format! Sure, you can write a program today, and if it's not too complex, you will be able to comprehend what you wrote a week from now. But, if it's complex, you may lose all track of what's taking place, not next week but within the next hour of programming.

In 1984, I began work on CBREEZE, a translator program that accepts BASIC language source code and converts it to C source code. It does this without using hidden functions, and it is aimed specifically at the student who is trying to learn C. To demonstrate the CBREEZE conversion output, consider the following BASIC source code.

```
10 FOR X%=0 TO 4000
20 PRINT X%
30 NEXT
40 PRINT"now is the time for all good men"
50 Y%=21
60 PRINT Y%*23
```

This simple program is converted to its exact equivalent C language source code by

CBREEZE, which outputs the following.

```
#include <stdio.h>
void main(void)
{

      int x, y;

      for (x = 0; x <= 4000; ++x)
                printf(" %d \n", x);

      printf("now is the time for all good men\n");

      y = 21;

      printf(" %d \n", y * 23);

}
```

This ANSI C program has not been touched by human hands. Maybe that's why it looks so clean! This is the exact output from CBREEZE, and it follows proper C language formatting to the letter. Now, if a program can do this, so can a human being, because it was a human being who wrote the translating program. The program is easy to understand and looks much better than the aberration that follows.

```
   #include<stdio.h>main(){intx,y;for(x=0;x<=40
00;++x)printf(" %d \n",x);printf("now is the time for
all good men\n");y= 21;printf(" %d \n", y * 23);}
```

This is an approximation of what the compiler sees.

At this point in a seminar, I'm usually asked if I followed these formatting conventions when writing CBREEZE over a two-year period of time, inputting more than 100,000 lines of source code. The answer is demonstrated in the following partial source code listing, which amounts to approximately one-third of the total source code of one of the smaller function definitions in the CBREEZE translator.

```
/* Copyright (c) 1984 R.J. Traister & Associates */
void araysort(char *qw)
{

      struct vfile *fp, *fdd, *vopen();
```

```
        FILE *fd;
        char a[250], b[260];
        int x, d, z, xyz;

        x = d = z = xyz = 0;

        fp = vopen(qw, "r");
        fdd = vopen("temp.ox", "w");

        while ((vgets(a, 200, fp)) != NULL)
              if ((index(a, "DIM ")) >= 0)
                  x = getout(a);

        vwind(fp);

        if ((fd = fopen("aray.xc", "r")) == NULL) {
            while ((vgets(a, 200, fp)) != NULL)
                   vputs(a, fdd);

        vclose(fp);
        vclose(fdd);
        return;
    }

        while ((vgets(a, 200, fp)) != NULL) {
               while ((fgets(b, 250, fd)) != NULL) {
                     replace(b, "\n", "");
                        while ((z = index(a, b)) >=
0 && (isalpha(a[z - 1])) == 0)
                                scarp(a, z);
                }

                rewind(fd);
                vputs(a, fdd);

        }

        unlink("aray.xc");
        vclose(fp);
        fclose(fd);
        vclose(fdd);

    }
```

To repeat, this is a portion of one of the smaller modules in the CBREEZE translator. The finished program consisted of hundreds of these modules and took years to complete. Can you see the importance of proper formatting? During the authoring of this software, I might have written one module in January and not referenced it again for six months. When it was time to return to this same module to make coding updates, it was often necessary to study what I had written six months prior to refresh my memory as to exactly what I had programmed and why! Can you imagine what agony I would have gone through had this module looked like the following example?

```
araysort(char *qw){
struct vfile *fp,*fdd,*vopen();
FILE *fd;
]char a[250],b[260];
int x,d,z,xyz;
x=d=z=xyz=0;
fp=vopen(qw, "r");
fdd=vopen("temp.ox","w");
while((vgets(a,200,fp))!=NULL)
if((index(a,"DIM "))>=0)
x=getout(a);
vwind(fp);
if((fd=fopen("aray.xc","r"))==NULL) {
while((vgets(a,200,fp))!=NULL)
vputs(a,fdd);
vclose(fp);
vclose(fdd);
return;
}
while((vgets(a,200,fp))!=NULL) {
while((fgets(b,250,fd))!=NULL) {
replace(b,"\n","");
while((z=index(a,b))>=0&&(isalpha(a[z-1]))==0)
scarp(a,z);
}
rewind(fd);
vputs(a,fdd);
}
      unlink("aray.xc");
vclose(fp);
fclose(fd);
vclose(fd);
..... and so forth - ad infinitum
```

If I had programmed in this manner, CBREEZE would not be on the market today. It would never have been completed. If you have never been involved in a commercial software programming environment, then you can't know that not understanding what you wrote yesterday is an ongoing ordeal. The complexities and variations involved in one project demand methods of deciphering what may have been written at 3 A.M. after 20 hours of straight programming effort some months after the fact!

Again someone will ask, "Why can't I develop my own formatting system, one that suits me best?" My first response to this is, "You are probably not qualified to do so." This always elicits an evil look from the person asking the question. More important, it is mandatory in commercial programming environments that others on your team be able to read what you wrote. If you have your own formatting code, and ten other team members have theirs, the system breaks down. You may not be working with a team now, but there is always that possibility in the future, so don't develop bad habits!

For me, the correct method of writing C source code, from the standpoint of formatting, has become automatic. I do not consider it a shortcut, for instance to delete the spaces between operators and their left/right values. To delete these spaces purposely would take me far longer than it does to put them in, such is the automatic formatting method that I have embraced during thousands of hours at the keyboard. You will grasp the concepts of C language pointers discussed in this book and every other phase of C language if you make yourself adhere to the programming conventions outlined here and garnered directly from *The C Programming Language* by Kernighan and Ritchie.

However, there are plenty of bad examples of C source code to influence beginners. Unfortunately, many of these less than ideal examples appear in documentation that accompanies some of the most popular compilers. "If the people who wrote the software do it this way, why shouldn't I?" is a legitimate question that I can only answer with, "Because it's WRONG!"

Incidentally, I'm sure a number of programmers will not agree with my very strong opinions on formatting source code, while others will. I will state for the record that, as a C language teacher, I am more than a little concerned about the bastardization of what is a beautiful source code format. ANSI C was developed to bring standardization to the C programming language. This is good, and I believe we need to carry this standard forward to source code formatting as well, not to preserve the grand old heritage of C language, doing it the way those first C pioneers did, but for the sake of clarity and understanding, allowing more programmers into the ranks.

As I was writing this book, a commercial software package arrived which I hoped to review for a magazine article. Believe it or not, the documentation was filled with source code examples that used left margin tactics, i.e., no indentations, no separators, no nothing! I sent it back.

Format for a Format

The following guidelines were used by the CBREEZE translator as rules of format. Perhaps they will aid you in formatting your own C programs.

Braces

Braces ({}) encompass a major control module. If that module is contained in a function, then the opening brace is positioned on the left margin immediately under the function name, as the example below illustrates.

```
char *strcpy(char *a, char *b)
{
```

When braces are used to confine operational blocks within a program, as would be the case with a loop containing more than one program statement, the opening brace follows the control expression on the same line and is separated by a single whitespace.

```
for (x = 0; x < 456; ++x) {
      printf("%d\n", x);
                        y +=5;
                  }
```

The closing brace is placed on the margin position that began the control expression and on a separate line immediately beneath the encapsulated statements.

Indentations

Indentations are always made in steps of five. If we assume that main() is located on the left hand margin (indent 0), then all other program statements are indented at locations 5, 10, 15, 20, etc. Assuming the prior *for* loop example begins at indent 5, then the statements within the loop begin at indent 10. Should there be a nested *for* loop within this one, then the *for* statement would be written at indent 10, and all program statements within this nested loop would be indented another 5 spaces (indent 15). The following program fragment shows this.

```
for (x = 0; x <= 465; ++x) {
      printf("%d\n", x);
      y += 5;
      for (z = 90; z > 0; –z) {
            strcpy(d, f);]
            ++g;
      }
}
```

This makes nested operations easier to differentiate.

Parentheses

Parentheses immediately follow a function name but are separated from *for*, *while*, *if*, and other statements by a space, as is shown below.

```
for (x = 0; x < 5; ++x)
     strcpy(a, b);
```

The *for* statement is separated from the opening paren by a space, whereas the strcpy() function is immediately followed by the paren.

Arithmetic Operators

Arithmetic operators are delineated by a space on the left and on the right. Other types of operators usually are not.

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

This statement clearly shows the use of arithmetic operators as opposed to other operator types. The expression below contains other types of operators along with those used for arithmetic purposes.

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

Here, the increment operator (++) is not as readily confused with the addition operator (+), and the multiplicative operator (*) is not confused with the indirection operator.

Commas

Commas are always followed by a whitespace when used as argument delineators.

```
add(a, b, c, d, e);
```

The same applies to semicolons used within *for*, *do*, and *while* statements. Note that commas and semicolons are not preceded by a whitespace when used in this manner.

Summary

The proper formatting of ANSI C source code is essential to gain a full understanding of the language. The difference between properly and improperly formatted code is like the difference between a neatly typed letter and one written in a messy scrawl.

By observing a few simple formatting rules, all source code will be easier to understand by you and by those who are interested in your programs. Of major importance is the fact that properly formatted programs will be more easily understood by the person who inputs the code lines, the programmer. This allows for quicker debugging of complex program modules and facilitates the addition of changes that are made to existing source code.

Index