C LANGUAGE TUTORIAL

This tutorial teaches the entire C programming language. It is composed of 13 chapters which should be studied in order since topics are introduced in a logical order and build upon topics introduced in previous chapters. It is to the students benefit to download the source code for the example programs, then compile and execute each program as it is studied. The diligent student will modify the example program in some way, then recompile and execute it to see if he understands the material studied for that program. This will provide the student with valuable experience using his compiler.

The recommended method of study is to print the text for one or two chapters, download the example programs, and study the material by loading the example programs in the compiler's editor for viewing. Following successful completion of each chapter, additional chapters can be downloaded as progress is made.

Version 2.8 - Sept 8, 1996 - (Files restructured on March 15, 1997)

This tutorial is distributed as shareware which means that you do not have to pay to use it. However, the author spent a good deal of time and financial resources to develop this tutorial and requests that you share in the financial burden in a very small way, but only if you felt the tutorial was valuable to you as an aid in learning to program in C. If you wish to remit a small payment to the author, full instructions for doing so will be given by clicking the link below. If you do not wish to remit any payment, please feel free to use the tutorial anyway. In either case, I hope you find programming in C to be rewarding and profitable. I personally think it is an excellent language.

Download the HTML Documentation - (chtm.zip) Download all of the above documents in one packed

file. This file (about 155k) contains the 14 files listed above (plus the diagrams) which can be downloaded and unpacked for use locally. The content of this file is identical to the content of the above files. There are no executable files in this group of files.

Download the Source Code - (csrc.zip) Download all example programs. This file (about 41k) contains 79 source files which are all explained in the 13 chapters of text. There are no executable files in this group of files.

Download the Answers to Exercises- (cans.zip) Download the authors answers to all of the programming exercises. This file (about 11k) contains 27 source files. There are no executable files in this group of files.

Download the pkunzip executable - (pkunzip.exe) Download pkunzip.exe version 2.04 to unzip the source code. This executable is pre-registered for your use in unzipping any Coronado Enterprises tutorial files. It will unpack and generate the zipped files in the current directory and all will be ASCII source code files. To unzip the source code files, execute the following DOS command;

```
pkunzip csrc.zip
```

Or, to unzip the answers to programming exercises, execute the following DOS command;

```
pkunzip cans.zip
```

Gordon Dodrill - dodrill@swcp.com - Please email any comments or suggestions.

Introduction to the C Tutorial

C IS USUALLY FIRST

The programming language C was originally developed by Dennis Ritchie of Bell Laboratories and was designed to run on a PDP-11 with a UNIX operating system. Although it was originally intended to run under UNIX, there has been a great interest in running it under the MS-DOS operating system on the IBM PC and compatibles. It is an excellent language for this environment because of the simplicity of expression, the compactness of the code, and the wide range of applicability. Also, due to the simplicity and ease of writing a C compiler, it is usually the first high level language available on any new computer, including microcomputers, minicomputers, and mainframes.

C is not the best beginning language because it is somewhat cryptic in nature. It allows the programmer a wide range of operations from high level down to a very low level, approaching the level of assembly language. There seems to be no limit to the flexibility available. One experienced C programmer made the statement, "You can program anything in C", and the statement is well supported by my own experience with the language. Along with the resulting freedom however, you take on a great deal of responsibility because it is very easy to write a program that destroys itself due to the silly little errors that a good Pascal compiler will flag and call a fatal error. In C, you are very much on your own as you will soon find.

I ASSUME YOU KNOW NOTHING ABOUT C

In order to successfully complete this tutorial, you will not need any prior knowlede of the C programming language. I will begin with the most basic concepts of C and take you up to the highest level of C programming including the usually intimidating concepts of pointers, structures, and dynamic allocation. To fully understand these concepts, it will take a good bit of time and work on your part because they are not particularly easy to grasp, but they are very powerful tools. Enough said about that, you will see their power when we get there, just don't allow yourself to worry about them yet.

Programming in C is a tremendous asset in those areas where you may want to use Assembly Language but would rather keep it a "simple to write" and "easy to maintain" program. It has been said that a program written in C will pay a premium of a 20 to 50% increase in runtime because no high level language is as compact or as fast as Assembly Language. However, the time saved in coding can be tremendous, making it the most desirable language for many programming chores. In addition, since most programs spend 90 percent of their operating time in only 10 percent or less of the code, it is possible to write a program in C, then rewrite a small portion of the code in Assembly Language and approach the execution speed of the same program if it were written entirely in Assembly Language.

Even though the C language enjoys a good record when programs are transported from one implementation to another, there are differences in compilers that you will find anytime you try to use another compiler. Most of the differences become apparent when you use nonstandard extensions such as calls to the DOS BIOS when using MS-DOS, but even these differences can be minimized by careful choice of programming constructs.

Throughout this tutorial, every attempt will be made to indicate to you what constructs are available in every C compiler because they are part of the ANSI-C standard, the accepted standard of C

programming.

WHAT IS THE ANSI-C STANDARD?

When it became evident that the C programming language was becoming a very popular language available on a wide range of computers, a group of concerned individuals met to propose a standard set of rules for the use of the C programming language. The group represented all sectors of the software industry and after many meetings, and many preliminary drafts, they finally wrote an acceptable standard for the C language. It has been accepted by the American National Standards Institute (ANSI), and by the International Standards Organization (ISO). It is not forced upon any group or user, but since it is so widely accepted, it would be economic suicide for any compiler writer to refuse to conform to the standard.

YOU MAY NEED A LITTLE HELP

Modern C compilers are very capable systems, but due to the tremendous versatility of a C compiler, it could be very difficult for you to learn how to use it effectively. If you are a complete novice to programming, you will probably find the installation instructions somewhat confusing. You may be able to find a colleague or friend that is knowledgeable about computers to aid you in setting up your compiler for initial use.

This tutorial cannot cover all aspects of programming in C, simply because there is too much to cover, but it will instruct you in all you need for the majority of your programming in C, and it will introduce essentially all of the C language. You will receive instruction in all of the programming constructs in C, but what must be omitted are methods of programming, since these can only be learned by experience. More importantly, it will teach you the vocabulary of C so that you can go on to more advanced techniques using the programming language C. A diligent effort on your part to study the material presented in this tutorial will result in a solid base of knowledge of the C programming language. You will then be able to intelligently read technical articles or other textbooks on C and greatly expand your knowledge of this modern and very popular programming language.

HOW TO USE THIS TUTORIAL

This tutorial is written in such a way that the student should sit before his computer and study each example program by displaying it on the monitor and reading the text which corresponds to that program. Following his study of each program, he should then compile and execute it and observe the results of execution with his compiler. This enables the student to gain experience using his compiler while he is learning the C programming language. It is strongly recommended that the student study each example program in the given sequence then write the programs suggested at the end of each chapter in order to gain experience in writing C programs.

THIS IS WRITTEN PRIMARILY FOR MS-DOS

This tutorial is written primarily for use on an IBM-PC or compatible computer but can be used with any ANSI standard compiler since it conforms so closely to the ANSI standard. In fact, a computer is not even required to study this material since the result of execution of each example program is given in comments at the end of each program.

RECOMMENDED READING AND REFERENCE MATERIAL

"The C Programming Language - Second Edition", Brian W. Kernigan & Dennis M. Ritchie, Prentice Hall, 1988

This is the definitive text of the C programming language and is required reading for every serious C programmer. Although the first edition was terse and difficult to read, the second edition is easier to read and extremely useful as both a learning resource and a reference guide.

Any ANSI-C textbook

Each student should posess a copy of a book that includes a definition of the entire ANSI-C specification and library. Go to a good bookstore and browse for one.

Return to Table of Contents

Advance to Chapter 1

Gordon Dodrill - dodrill@swcp.com - Please email any comments or suggestions.

C Tutorial - Chapter 1

GETTING STARTED

WHAT IS AN IDENTIFIER?

Before you can do anything in any language, you must know how to name an identifier. An identifier is used for any variable, function, data definition, etc. In the C programming language, an identifier is a combination of alphanumeric characters, the first being a letter of the alphabet or an underline, and the remaining being any letter of the alphabet, any numeric digit, or the underline.

Two rules must be kept in mind when naming identifiers.

1. The case of alphabetic characters is significant. Using **INDEX** for a variable name is not the same as using **index** and neither of them is the same as using **InDeX** for a variable name. All three refer to different variables.
2. According to the ANSI-C standard, at least 31 significant characters can be used and will be considered significant by a conforming ANSI-C compiler. If more than 31 are used, all characters beyond the 31st may be ignored by any given compiler.

WHAT ABOUT THE UNDERLINE?

The underline can be used as part of a variable name, and adds greatly to the readability of the resulting code. It is used by some, but not all, experienced C programmers. A few underlines are used for illustration in this tutorial. Since most compiler writers use the underline as the first character for variable names internal to the system, you should refrain from using the underline to begin an identifier to avoid the possibility of a name clash. To get specific, identifiers with two leading underscores are reserved for the compiler as well as identifiers beginning with a single underscore and using an upper case alphabetic character for the second. If you make it a point of style to never use an identifier with a leading underline, you will not have a naming clash with the system.

It adds greatly to the readability of a program to use descriptive names for variables and it would be to your advantage to do so. Pascal and Ada programmers tend to use long descriptive names, but most C programmers tend to use short cryptic names. Most of the example programs in this tutorial use very short names for that reason, but a few longer names are used for illustrative purposes.

KEYWORDS

There are 32 words defined as keywords in C. These have predefined uses and cannot be used for any other purpose in a C program. They are used by the compiler as an aid to compiling the program. They are always written in lower case. A complete list follows;

```
auto      double    int       struct
break     else      long      switch
case      enum      register  typedef
char      extern    return    union
const     float     short     unsigned
continue  for       signed    void
default   goto      sizeof    volatile
```

```
do          if          static      while
```

In addition to this list of keywords, your compiler may define a few more. If it does, they will be listed in the documentation that came with your compiler. Each of the above keywords will be defined, illustrated, and used in this tutorial.

WE NEED DATA AND A PROGRAM

Any computer program has two entities to consider, the data, and the program. They are highly dependent on one another and careful planning of both will lead to a well planned and well written program. Unfortunately, it is not possible to study either completely without a good working knowledge of the other. For that reason, this tutorial will jump back and forth between teaching methods of program writing and methods of data definition. Simply follow along and you will have a good understanding of both. Keep in mind that, even though it seems expedient to sometimes jump right into coding the program, time spent planning the data structures will be well spent and the quality of the final program will reflect the original planning.

HOW THIS TUTORIAL IS WRITTEN

As you go through the example programs, you will find that every program is complete. There are no program fragments that could be confusing. This allows you to see every requirement that is needed to use any of the features of C as they are presented. Some tutorials I have seen give very few, and very complex examples. They really serve more to confuse the student. This tutorial is the complete opposite because it strives to cover each new aspect of programming in as simple a context as possible.

Throughout this tutorial, **keywords, variable names,** and **function names** will be given in boldface as an aid to clarity. These terms will be completely defined throughout the tutorial.

RESULT OF EXECUTION

The result of executing each program will be given in comments at the end of the program listing after the comment is defined in about the fourth program of chapter 2. If you feel confident that you completely understand the program, you can simply refer to the result of execution to see if you understand the result. In this case, it will not be necessary for you to compile and execute every program. It would be a good exercise for you to compile and execute some of them however, because all C compilers will not generate exactly the same results and you need to get familiar with your own compiler.

Example program ------> **FIRSTEX.C**

At this point, you should compile and execute FIRSTEX.C if you have not yet done so, to see that your C compiler is properly loaded and operating. Don't worry about what the program does yet. In due time you will understand it completely.

Note that this program will compile and execute properly with any good compiler.

A WORD ABOUT COMPILERS

All of the example programs in this tutorial will compile and execute correctly with any good ANSI compatible C compiler. Some compilers have gotten extremely complex and hard to use for a beginning

C programmer, and some only compile and build Microsoft Windows programs. Fortunately, most of the C compilers available have a means of compiling a standard C program which is written for the DOS environment and includes none of the Windows extensions. You should check your documentation for the capabilities and limitations of your compiler. If you have not yet purchased a C compiler, you should find one that is ANSI-C compliant, and that also has the ability to generate a DOS executable if you are planning to use the DOS operating system.

ANSWERS TO PROGRAMMING EXERCISES

There are programming exercises at the end of most of the chapters. You should attempt to do original work on each of the exercises before referring to the answers (all of which are zipped into cans.zip) in order to gain your own programming experience. These answers are given for your information in case you are completely stuck on how to solve a particular problem. These answers are not meant to be the only answer, since there are many ways to program anything, but they are meant to illustrate one way to solve the suggested programming problem.

The answers are all in source files named in the format CHnn_m.C where nn is the chapter number, and m is the exercise number. If more than one answer is required, an A, B, or C is included following the exercise number.

Return to Table of Contents

Advance to Chapter 2

Gordon Dodrill - dodrill@swcp.com - Please email any comments or suggestions.

C Tutorial - Chapter 2

GETTING STARTED IN C

YOUR FIRST C PROGRAM

Example program ------> **TRIVIAL.C**

The best way to get started with C is to actually study a program, so load the file named TRIVIAL.C and display it on the monitor. You are looking at the simplest possible C program. There is no way to simplify this program or to leave anything out. Unfortunately, the program doesn't do anything.

The word **main** is very important, and must appear once, and only once in every C program. This is the point where execution is begun when the program is run. We will see later that this does not have to be the first statement in the program but it must exist as the entry point. Following the **main** program name is a pair of parentheses which are an indication to the compiler that this is a function. We will cover exactly what a function is in due time. For now, I suggest that you simply include the pair of parentheses.

The two curly brackets in lines 2 and 3, properly called braces, are used to define the limits of the program itself. The actual program statements go between the two braces and in this case, there are no statements because the program does absolutely nothing. You can compile and run this program, but since it has no executable statements, it does nothing. Keep in mind, however, that it is a valid C program. When you compile this program, you may get a warning. You can ignore the warning and we will discuss it later in this tutorial, or you can modify the program so that it appears as follows;

```
int main()
{
   return 0;
}
```

This modified program must compile on any good C compiler since it conforms to the ANSI-C standard. We will explain the difference in these two programs later in this tutorial.

A PROGRAM THAT DOES SOMETHING

Example program ------> **WRTSOME.C**

For a much more interesting program, load the program named WRTSOME.C and display it on your monitor. It is the same as the previous program except that it has one executable statement between the braces plus the obligatory **return** statement.

The executable statement is a call to a function supplied as a part of your C library. Once again, we will not worry about what a function is, but only how to use this one named **printf()**. In order to output text to the monitor, the desired text is put within the function parentheses and bounded by quotation marks. The end result is that whatever text is included between the quotation marks will be displayed on the monitor when the program is run.

Notice the semi-colon at the end of line 5. C uses a semi-colon as a statement terminator, so the semi-colon is required as a signal to the compiler that this line is complete. This program is also

executable, so you can compile and run it to see if it does what you think it should. It should cause the text between the quotation marks to appear on the monitor when you execute it.

You can ignore the statements in lines 1 and 7 in this program and similar statements in each of the remaining programs in this chapter. These will be fully described later in this tutorial. We will also define why the word **int** is used at the begining of line 3. We have a few preliminary topics to cover before we get to these items.

ANOTHER PROGRAM WITH MORE OUTPUT

Example program ------> **WRTMORE.C**

Load the program WRTMORE.C and display it on your monitor for an example with more output and another small but important concept. You will see that there are four executable statements in this program, each one being a call to the function **printf().** The top line will be executed first, then the next, and so on, until the fourth line is complete. The statements are executed sequentially from top to bottom.

Notice the funny character near the end of the first line, namely the backslash. The backslash is used in the **printf()** statement to indicate that a special control character is following. In this case, the "**n**" indicates that a newline is requested. This is an indication to return the cursor to the left side of the monitor and move down one line. Any place within printed text that you desire, you can put a newline character to start a new line. You could even put it in the middle of a word and split the word between two lines.

A complete description of this program is now possible. The first **printf()** outputs a line of text and returns the carriage. (Of course, there is no carriage, but the cursor is moved to the next line on the monitor. The terminology carries over from the days of teletypes.) The second **printf()** outputs a line of text but does not return the carriage so that the third line is appended to the end of the second, then followed by two carriage returns, resulting in a blank line. Finally the fourth **printf()** outputs a line followed by a carriage return and the program is complete.

After compiling and executing WRTMORE.C, the following text should be displayed on your monitor;

```
This is a line of text to output.
And this is another line of text.

This is a third line.
```

Compile and execute this program to see if it gives you this output. It would be a good idea at this time for you to experiment by adding additional lines of printout to see if you understand how the statements really work. Add a few carriage returns in the middle of a line to prove to yourself that it works as stated, then compile and execute the modified program. The more you modify and compile the example programs included with this tutorial, the more you will learn as you work your way through it.

LET'S PRINT SOME NUMBERS

Example program ------> **ONEINT.C**

Load the file named ONEINT.C and display it on the monitor for our first example of how to work with data in a C program. The entry point **main()** should be clear to you by now as well as the beginning

brace. The first new thing we encounter is line 5 containing `int index;` which is used to define an integer variable named **index**. The word **int** is a keyword in C, and can not be used for anything else. It defines a variable that can store a whole number within a predefined range of values. We will define an actual range later. The variable name, **index**, can be any name that follows the rules for an identifier and is not one of the keywords for C. The final character on the line, the semi-colon, is the statement terminator as discussed earlier.

Note that, even though we have defined a variable, we have not yet assigned a value to it, so it contains an undefined value. We will see in a later chapter that additional integers could also be defined on the same line, but we will not complicate the present situation.

Observing the main body of the program, you will notice that there are three statements that assign a value to the variable **index**, but only one at a time. The statement in line 7 assigns the value of 13 to **index**, and its value is printed out by line 8. (We will see how shortly. Trust me for the time being.) Later, the value of 27 is assigned to **index**, and finally 10 is assigned to it, each value being printed out. It should be intuitively clear that **index** is indeed a variable and can store many different values but only one value at a time of course.

Please note that many times the words "printed out" are used to mean "displayed on the monitor". You will find that in many cases experienced programmers take this liberty, probably due to the **printf()** function being used for monitor display.

HOW DO WE PRINT NUMBERS?

To keep our promise, let's return to the **printf()** statements for a definition of how they work. Notice that they are all identical and that they all begin just like the **printf()** statements we have seen before. The first difference occurs when we come to the % character. This is a special character that signals the output routine to stop copying characters to the output and do something different, usually to output the value of a variable. The **%** sign is used to signal the output of many different types of variables, but we will restrict ourselves to only one for this example. The character following the **%** sign is a **d**, which signals the output routine to get a decimal value and output it. Where the decimal value comes from will be covered shortly. After the **d**, we find the familiar **\n**, which is a signal to return the video "carriage", and the closing quotation mark.

All of the characters between the quotation marks define the pattern of data to be output by this statement. Following the output pattern, there is a comma followed by the variable name **index**. This is where the **printf()** statement gets the decimal value which it will output because of the **%d** we saw earlier. The system substitutes the current value of the variable named **index** for the **%d** and copies it to the monitor. We could add more **%d** output field descriptors anywhere within the brackets and more variables following the description to cause more data to be printed with one statement. Keep in mind however, that the number of field descriptors and the number of variable definitions must be the same or the runtime system will generate something we are not expecting.

Much more will be covered at a later time on all aspects of input and output formatting. A reasonably good grasp of these fundamentals are necessary in order to understand the following lessons. It is not necessary to understand everything about output formatting at this time, only a fair understanding of the basics.

Compile and run ONEINT.C and observe the output. Two programming exercises at the end of this chapter are based on this program.

HOW DO WE ADD COMMENTS IN C?

Example program ------> **COMMENTS.C**

Load the file named COMMENTS.C and observe it on your monitor for an example of how comments can be added to a C program. Comments are added to make a program more readable to you but represent nonsense to the compiler, so we must tell the compiler to ignore the comments completely by bracketing them with special characters. The slash star combination is used in C for comment delimiters, and are illustrated in the program at hand. Please note that the program does not illustrate good commenting practice, but is intended to illustrate where comments can go in a program. It is a very sloppy looking program.

The slash star combination in line 3 introduces the first comment and the star slash at the end of that line terminates this comment. Note that this comment is prior to the beginning of the program illustrating that a comment can precede the program itself. Good programming practice would include a comment prior to the program with a short introductory description of the program. The comment in line 5 is after the main program entry point and prior to the opening brace for the program code itself.

The third comment starts after the first executable statement in line 7 and continues for four lines. This is perfectly legal because a comment can continue for as many lines as desired until it is terminated. Note carefully that if anything were included in the blank spaces to the left of the three continuation lines of the comment, it would be part of the comment and would not be compiled, but totally ignored by the compiler. The last comment, in line 15, is located following the completion of the program, illustrating that comments can go nearly anywhere in a C program.

Experiment with this program by adding comments in other places to see what will happen. Comment out one of the **printf()** statements by putting comment delimiters both before and after it and see that it does not get executed and therefore does not produce a line of printout.

Comments are very important in any programming language because you will soon forget what you did and why you did it. It will be much easier to modify or fix a well commented program a year from now than one with few or no comments. You will very quickly develop your own personal style of commenting.

Some C compilers will allow you to "nest" comments which can be very handy if you need to "comment out" a section of code during debugging. Since nested comments are not a part of the ANSI-C standard, none will be used in this tutorial. Check the documentation for your compiler to see if they are permitted with your implementation of C. Even though they may be allowed, it is a good idea to refrain from their use, since they are rarely used by experienced C programmers, and using them may make it difficult to port your code to another compiler if the need should arise.

GOOD FORMATTING STYLE

Example program ------> **GOODFORM.C**

Load the file GOODFORM.C and observe it on your monitor. It is an example of a well formatted program. Even though it is very short and therefore does very little, it is very easy to see at a glance what

it does. With the experience you have already gained in this tutorial, you should be able to very quickly grasp the meaning of the program in it's entirety. Your C compiler ignores all extra spaces and all carriage returns giving you considerable freedom in formatting your program. Indenting and adding spaces is entirely up to you and is a matter of personal taste. Compile and run the program to see if it does what you expect it to do.

Example program ------> **UGLYFORM.C**

Now load and display the program UGLYFORM.C and observe it. How long will it take you to figure out what this program will do? It doesn't matter to the compiler which format style you use, but it will matter to you when you try to debug your program. Compile this program and run it. You may be surprised to find that it is the same program as the last one, except for the formatting. Don't get too worried about formatting style yet. You will have plenty of time to develop a style of your own as you learn the C language. Be observant of styles as you see C programs in magazines and books.

This covers some of the basic concepts of programming in C, but as there are many other things to learn, we will forge ahead to additional program structure. It will definitely be to your advantage to do the programming exercises at the end of each chapter. They are designed to augment your studies and teach you to use your compiler.

PROGRAMMING EXERCISES

1. Write a program to display your name on the monitor.
2. Modify the program to display your address and phone number on separate lines by adding two additional **printf()** statements.
3. Remove line 7 from ONEINT.C by commenting it out, then compile and execute the resulting program to see the value of an uninitialized variable. This can be any value within the allowable range for that variable. If it happens to have the value of zero, that is only a coincidence, but then zero is the most probable value to be in an uninitialized variable because there are lots of zero values floating around in a computer's memory. It is actually legal for the program to abort if you refer to a variable that you failed to initialize, but few compilers, if any, will actually do so.
4. Add the following two lines just after the last **printf()** of ONEINT.C to see what it does. Study it long enough to completely understand the result.

```
printf("Index is %d\n it still is %d\n it is %d",
                              index, index, index);
```

Return to Table of Contents

Advance to Chapter 3

Gordon Dodrill - dodrill@swcp.com - Please email any comments or suggestions.

C Tutorial - Chapter 3

PROGRAM CONTROL

THE WHILE LOOP

The C programming language has several structures for looping and conditional branching. We will cover them all in this chapter and we will begin with the **while** loop.

The **while** loop continues to loop while some condition is true. When the condition becomes false, the looping is discontinued. It therefore does just what it says it does, the name of the loop being very descriptive.

Example program ------> **WHILE.C**

Load the program WHILE.C and display it for an example of a **while** loop. We begin with a comment and the program entry point **main()**, then go on to define an integer variable named **count** within the body of the program. The variable is set to zero and we come to the **while** loop itself. The syntax of a **while** loop is just as shown here. The keyword **while** is followed by an expression of something in parentheses, followed by a compound statement bracketed by braces. As long as the expression in the parenthesis is true, all statements within the braces will be repeatedly executed. In this case, since the variable **count** is incremented by one every time the statements are executed, it will eventually reach 6. At that time the statement will not be executed because **count** is not less than 6, and the loop will be terminated. The program control will resume at the statement following the statements in braces.

We will cover the compare expression, the one in parentheses, in the next chapter. Until then, simply accept the expressions for what you think they should do and you will be correct for these simple cases.

Several things must be pointed out regarding the **while** loop. First, if the variable **count** were initially set to any number greater than 5, the statements within the loop would not be executed at all, so it is possible to have a **while** loop that never is executed. Secondly, if the variable were not incremented in the loop, then in this case, the loop would never terminate, and the program would never complete. Finally, if there is only one statement to be executed within the loop, it does not need delimiting braces but can stand alone.

Compile and run this program after you have studied it enough to assure yourself that you understand its operation completely. Note that the result of execution is given for this program, (and will be given for all of the remaining example programs in this tutorial) so you do not need to compile and execute every program to see the results. Be sure to compile and execute some of the programs however, to gain experience with your compiler.

You should make some modifications to any programs that are not completely clear to you and compile them until you understand them completely. The best way to learn is to try various modifications yourself.

We will continue to ignore the **#include** statement and the **return** statement in the example programs in this chapter. We will define them completely later in this tutorial.

THE DO-WHILE LOOP

Example program ------> **DOWHILE.C**

A variation of the **while** loop is illustrated in the program DOWHILE.C, which you should load and display. This program is nearly identical to the last one except that the loop begins with the keyword **do**, followed by a compound statement in braces, then the keyword **while**, and finally an expression in parentheses. The statements in the braces are executed repeatedly as long as the expression in the parentheses is true. When the expression in parentheses becomes false, execution is terminated, and control passes to the statements following this statement.

Several things must be pointed out regarding the **do-while** loop. Since the test is done at the end of the loop, the statements in the braces will always be executed at least once. Secondly, if the variable **i** were not changed within the loop, the loop would never terminate, and hence the program would never terminate.

It should come as no surprise to you that these loops can be nested. That is, one loop can be included within the compound statement of another loop, and the nesting level has no limit. This will be illustrated later.

Compile and run this program to see if it does what you think it should do.

THE FOR LOOP

Example program ------> **FORLOOP.C**

Load and display the file named FORLOOP.C on your monitor for an example of a program with a **for** loop. The **for** loop consists of the keyword **for** followed by a rather large expression in parentheses. This expression is really composed of three fields separated by semi-colons. The first field contains the expression `"index = 0"` and is an initializing field. Any expressions in this field are executed prior to the first pass through the loop. There is essentially no limit as to what can go here, but good programming practice would require it to be kept simple. Several initializing statements can be placed in this field, separated by commas.

The second field, in this case containing `"index < 6"`, is the test which is done at the beginning of each pass through the loop. It can be any expression which will evaluate to a true or false. (More will be said about the actual value of true and false in the next chapter.)

The expression contained in the third field is executed each time the loop is exercised but it is not executed until after those statements in the main body of the loop are executed. This field, like the first, can also be composed of several operations separated by commas.

Following the **for()** expression is any single or compound statement which will be executed as the body of the loop. A compound statement is any group of valid C statements enclosed in braces. In nearly any context in C, a simple statement can be replaced by a compound statement that will be treated as if it were a single statement as far as program control goes. Compile and run this program.

The **while** is convenient to use for a loop when you don't have any idea how many times the loop will be executed, and the **for** loop is usually used in those cases when you are doing a fixed number of iterations. The **for** loop is also convenient because it moves all of the control information for a loop into one place, between the parentheses, rather than at both ends of the code. It is your choice as to which you would rather use. Depending on how they are used, it is possible with each of these two loops to never execute

the code within the loop at all. This is because the test is done at the beginning of the loop, and the test may fail during the first iteration. The **do-while** loop however, due to the fact that the code within the loop is executed prior to the test, will always execute the code at least once.

THE IF STATEMENT

Example program ------> **IFELSE.C**

Load and display the file IFELSE.C for an example of our first conditional branching statement, the **if**. Notice first, that there is a **for** loop with a compound statement as its executable part containing two **if** statements. This is an example of how statements can be nested. It should be clear to you that each of the **if** statements will be executed 10 times.

Consider the first **if** statement. It starts with the keyword **if** followed by an expression in parentheses. If the expression is evaluated and found to be true, the single statement following the **if** is executed, and if false, the following statement is skipped. Here too, the single statement can be replaced by a compound statement composed of several statements bounded by braces. The expression `"data == 2"` is simply asking if the value of **data** is equal to 2. This will be explained in detail in the next chapter. (Simply suffice for now that if `"data = 2"` were used in this context, it would mean a completely different thing. You must use the double equal sign for comparing values.)

NOW FOR THE IF-ELSE

The second **if** is similar to the first with the addition of a new keyword, the **else** in line 17. This simply says that if the expression in the parentheses evaluates as true, the first expression is executed, otherwise the expression following the **else** is executed. Thus, one of the two expressions will always be executed, whereas in the first example the single expression was either executed or skipped. Both will find many uses in your C programming efforts. Compile and run this program to see if it does what you expect.

THE BREAK AND CONTINUE

Example program ------> **BREAKCON.C**

Load the file named BREAKCON.C for an example of two new statements. Notice that in the first **for** loop, there is an **if** statement that calls a **break** if **xx** equals 8. The **break** will jump out of the loop you are in and begin executing statements immediately following the loop, effectively terminating the loop. This is a valuable statement when you need to jump out of a loop depending on the value of some results calculated in the loop. In this case, when **xx** reaches the value of 8, the loop is terminated and the last value printed will be the previous value, namely 7. The **break** always jumps out of the loop just past the terminating brace.

The next **for** loop starting in line 15, contains a **continue** statement which does not cause termination of the loop but jumps out of the present iteration. When the value of **xx** reaches 8 in this case, the program will jump to the end of the loop and continue executing the loop, effectively eliminating the **printf()** statement during the pass through the loop when **xx** is eight. The **continue** statement always jumps to the end of the loop just prior to the terminating brace. At that time the loop is terminated or continues based on the result of the loop test.

Be sure to compile and execute this program.

THE SWITCH STATEMENT

Example program ------> **SWITCH.C**

Load and display the file SWITCH.C for an example of the biggest construct yet in the C language, the **switch**. The **switch** is not difficult, so don't let it intimidate you. It begins with the keyword **switch** followed by a variable in parentheses which is the switching variable, in this case **truck**. As many cases as needed are then enclosed within a pair of braces. The reserved word **case** is used to begin each case, followed by the value of the variable for that case, then a colon, and the statements to be executed.

In this example, if the variable named **truck** contains the value 3 during this pass of the **switch** statement, the **printf()** in line 13 will cause `"The value is three\n"` to be displayed, and the **break** statement will cause us to jump out of the **switch**. The **break** statement here works in much the same manner as the loop, it jumps out just past the closing brace.

Once an entry point is found, statements will be executed until a **break** is found or until the program drops through the bottom of the **switch** braces. If the variable **truck** has the value 5, the statements will begin executing at line 17 where `"case 5 :"` is found, but the first statements found are where the case 8 statements are. These are executed and the **break** statement in line 21 will direct the execution out of the bottom of the switch just past the closing brace. The various case values can be in any order and if a value is not found, the default portion of the switch will be executed.

It should be clear that any of the above constructs can be nested within each other or placed in succession, depending on the needs of the particular programming project at hand. Note that the **switch** is not used as frequently as the loop and the **if** statements. In fact, the **switch** is used infrequently but should be completely understood by the serious C programmer. Be sure to compile and run SWITCH.C and examine the results.

THE EVIL GOTO STATEMENT

Example program ------> **GOTOEX.C**

Load and display the file GOTOEX.C for an example of a file with some **goto** statements in it. To use a **goto** statement, you simply use the reserved word **goto** followed by the symbolic name to which you wish to jump. The name is then placed anywhere in the program followed by a colon. You can jump nearly anywhere within a function, but you are not permitted to jump into a loop, although you are allowed to jump out of a loop.

This particular program is really a mess but it is a good example of why software writers are trying to eliminate the use of the **goto** statement as much as possible. The only place in this program where it is reasonable to use the **goto** is the one in line 23 where the program jumps out of the three nested loops in one jump. In this case it would be rather messy to set up a variable and jump successively out of each of the three nested loops but one **goto** statement gets you out of all three in a very concise manner.

Some persons say the **goto** statement should never be used under any circumstances, but this is narrow minded thinking. If there is a place where a **goto** will clearly do a neater control flow than some other construct, feel free to use it. It should not be abused however, as it is in the rest of the program on your monitor.

Entire books are written on "gotoless" programming, better known as Structured Programming.

Compile and run GOTOEX.C and study its output. It would be a good exercise to rewrite it and see how much more readable it is when the statements are listed in order.

FINALLY, A MEANINGFUL PROGRAM

Example program ------> **TEMPCONV.C**

Load the file named TEMPCONV.C for an example of a useful, even though somewhat limited program. This is a program that generates a list of centigrade and fahrenheit temperatures and prints a message out at the freezing point of water and another at the boiling point of water.

Of particular importance is the formatting. The header is several lines of comments describing what the program does in a manner that catches the readers attention and is still pleasing to the eye. You will eventually develop your own formatting style, but this is a good way to start. Also if you observe the **for** loop, you will notice that all of the contents of the compound statement are indented 3 spaces to the right of the **for** keyword, and the opening and closing braces are lined up under the "f" in **for**. This makes debugging a bit easier because the construction becomes very obvious. (The next example program will illustrate two additional methods of formatting braces.) You will also notice that the **printf()** statements that are in the **if** statements within the big **for** loop are indented three additional spaces because they are part of yet another construct.

This is the first program in which we used more than one variable. The three variables are simply defined on three different lines and are used in the same manner as a single variable was used in previous programs. By defining them on different lines, we have an opportunity to define each with a comment. It would be possible to define them on one line, but to do so would remove the ability to include a comment on each line. This is illustrated in the next program. Be sure to compile and execute the current program.

ANOTHER POOR PROGRAMMING EXAMPLE

Example program ------> **DUMBCONV.C**

Recalling UGLYFORM.C from the last chapter, you saw a very poorly formatted program. If you load and display DUMBCONV.C you will have an example of poor formatting which is much closer to what you will find in practice. This is the same program as TEMPCONV.C with the comments removed and the variable names changed to remove the descriptive aspect of the names. Although this program does exactly the same as the last one, it is much more difficult to read and understand. You should begin to develop good programming practices now by studying this program to learn what not to do.

It would be beneficial for you to remove the indentation from the last two example programs to see how much more difficult it is to understand the structure of the program without the indentations.

OUR FIRST STYLE PROGRAM

Example program ------> **STYLE1.C**

This program does nothing practical except to illustrate various styles of formatting and how to combine some of the constructs introduced in this chapter. There is nothing in this program that we have not studied so far in this tutorial. The program is heavily commented and should be studied in detail by the

diligent C student to begin learning proper C programming style. Like all other example programs, this one can be compiled and executed, and should be.

PROGRAMMING EXERCISES

1. Write a program that writes your name on the monitor ten times. Write this program three times, once with each looping method.
2. Write a program that counts from one to ten, prints the values on a separate line for each, and includes a message of your choice when the count is 3 and a different message when the count is 7.

Return to Table of Contents

Advance to Chapter 4

Gordon Dodrill - dodrill@swcp.com - Please email any comments or suggestions.

C Tutorial - Chapter 4

ASSIGNMENT & LOGICAL COMPARES

Throughout this chapter, references are given to various ranges of variables. This refers to the range of values that can be stored in any given variable. Your compiler may use a different range for some of the variables since the ANSI standard does not define specific limits for all data types. Consult the documentation for your compiler for the exact range of each of the variable types.

INTEGER ASSIGNMENT STATEMENTS

Example program ------> **INTASIGN.C**

Load the file named INTASIGN.C and display it for an example of assignment statements. Three variables are defined for use in the program and the remainder of the program is merely a series of illustrations of various kinds of assignment statements. All three variables are defined on one line and have unknown values stored in them initially.

The first two lines of the assignment statements, lines 8 and 10, assign numerical values to the variables named **a** and **b**, and the next five lines illustrate the five basic arithmetic functions and how to use them. The fifth is the modulo operator and gives the remainder if the two variables were divided. It can only be applied to integral type variables, which will be defined later. Lines 15 and 16 illustrate how to combine some of the variables in relatively complex math expressions. All of the above examples should require no comment except to say that none of the equations are meant to be particularly useful except as illustrations.

Precedence of operators is a very important topic that you will need to study in detail at some point, but for now we will only need a few rules. When you have mixed arithmetic expressions, the multiplication and division operations are completed before the addition and subtraction operations when they are all at the same logical level. Therefore when evaluating **a * b + c / d**, the multiplication and division are done first, then the addition is performed. However in the expression **a * (b + c / d)**, the addition follows the division, but preceeds the multiplication because the operations are at two different logical levels as defined by the parentheses.

The expressions in lines 17 and 18 are perfectly acceptable as given, but we will see later in this chapter that there is another way to write these for more compact code.

VERY STRANGE LOOKING CODE

This brings us to lines 20 and 21 which may appear to you as being very strange. The C compiler scans the assignment statement from right to left, (which may seem a bit odd since we do not read that way), resulting in a very useful construct, namely the one given here. The compiler finds the value 20, assigns it to **c**, then continues to the left finding that the latest result of a calculation should be assigned to **b**. Thinking that the latest calculation resulted in a 20, it assigns that value to **b** also, and continues the leftward scan assigning the value 20 to **a** also. This is a very useful construct when you are initializing a group of variables. The statement in line 21 illustrates that it is possible to actually do some calculations to arrive at the value which will be assigned to all three variables. The values of **a**, **b**, and **c**, prior to the beginning of the statement in line 21 are used to calculate a value, which is then assigned to each of the

three variables.

As an aid to understanding, line 23 is given which contains parentheses to group the terms together in a meaningful way. Lines 20 and 23 are identical statements.

The program has no output, so compiling and executing this program will be very uninteresting. Since you have already learned how to display some integer results using the **printf()** function, it would be to your advantage to add some output statements to this program to see if the various statements do what you think they should do. You will need to add **#include <stdio.h>** to the beginning of the program if you are going to add **printf()** statements to the program.

You can add your own assignment statements also to gain experience with them.

DEFINITIONS FIRST THEN EXECUTABLE STATEMENTS

This would be a good time for a preliminary definition of a rule to be followed in C. The variable definitions are always given before any executable statements in any program block. This is why the variables are defined at the beginning of a block in this program and in every C program. If you try to define a new variable after some executable statements, your compiler will issue an error. A program block is any unit of one or more statements surrounded by braces. Actually, the block can even be empty but then there is no real need for it, except as a placeholder in early phases of code development. More will be said about blocks later.

ADDITIONAL DATA TYPES

Example program ------> **MORTYPES.C**

Loading and editing MORTYPES.C will illustrate how some additional data types can be used. Once again we have defined a few integer type variables which you should be fairly familiar with by now, but we have added two new types, the **char**, and the **float**.

The **char** type of data is nearly the same as the integer except that it can only be assigned numerical values between -128 and 127 on most microcomputer implementations of C, since it is usually stored in one byte of memory. Some implementations of C use a larger memory element for a **char** and will therefore cover a wider range of usable values. The **char** type of data is usually used for ASCII data, more commonly known as text. The text you are reading was originally written on a computer with a word processor that stored the words in the computer one character per byte. In contrast, the **int** data type is stored in two bytes of computer memory on nearly all microcomputers, but can be larger on some machines. In fact, most modern microcomputers are 32 bit machines that store an **int** in four bytes.

Keep in mind that, even though the **char** type variable was designed to hold a representation of an ASCII character, it can be used very effectively to store a very small value if desired. Much more will be discussed on this topic in chapter 7 when we discuss strings.

DATA TYPE MIXING

It would be profitable at this time to discuss the way C handles the two types **char** and **int**. Most operations in C that are designed to operate with integer type variables will work equally well with character type variables because they are an integral variable, which means that they have no fractional part. Those operations, when called on to use a **char** type variable, will actually promote the **char** data

into integer data before using it. For this reason, it is possible to mix **char** and **int** type variables in nearly any way you desire. The compiler will not get confused, but you might. It is good not to rely on this too much, but to carefully use only the proper types of data where they should be used.

The second new data type is the **float** type of data, commonly called floating point data. This is a data type which usually has a very large range, a relatively large number of significant digits, and a large number of computer words are required to store it. The **float** data type has a decimal point associated with it and several bytes of memory are required to store a single **float** type variable.

HOW TO USE THE NEW DATA TYPES

The first three lines of the program assign values to all nine of the defined variables so we can manipulate some of the data between the different types.

Since, as mentioned above, a **char** data type is in reality an integral data type which is automatically promoted to **int** when necessary, no special considerations need be taken to promote a **char** to an **int**, and a **char** type data field can be assigned to an **int** variable. When going the other way, an **int** type variable can be assigned to a **char** type variable and will translate correctly to a **char** type variable if the value is within the range of the **char**, possibly -128 to 127. If the value is outside of the range of **char**, most C compilers simply truncate the most significant bits and use the least significant bits.

Line 16 illustrates the simplicity of translating an **int** into a **float**. Simply assign it the new value and the system will do the proper conversion. When converting from **float** to **int** however, there is an added complication. Since there may be a fractional part of the floating point number, the system must decide what to do with it. By definition, it will truncate it and throw away the fractional part.

This program produces no output, and we haven't covered a way to print out **char** and **float** type variables, so you can't really get in to this program and play with the results. The next program will cover these topics for you.

Be sure to compile and run this program after you are sure you understand it completely. Note that the compiler may issue warnings about type conversions when compiling this program. They can be ignored because of the small values we are using to illustrate the various type conversions.

SOME TYPICAL SIZES

This list gives you some typical values for the various types available in C. Your compiler may offer different limits and sizes since there is a lot of latitude in what a compiler may offer. The values in this list are for Microsoft Visual C++ version 1.5 (16 bits) and Visual C++ version 2.0 (32 bits).

```
Type Name         Bytes   Range
  ------------- 16 bit system -------------
char              1       -128 to 127
signed char       1       -128 to 127
unsigned char     1       0 to 255
short             2       -32,768 to 32,767
unsigned short    2       0 to 65,535
int               2       -32,768 to 32,767
unsigned int      2       0 to 65,535
long              4       -2,147,483,648 to 2,147,483,647
```

```
unsigned long     4       0 to 4,294,967,295
float             4       3.4E+/-38 (7 digits)
double            8       1.7E+/-308 (15 digits)
long double      10       1.2E+/-4932 (19 digits)

  ------------- 32 bit system -------------
char              1       -128 to 127
signed char       1       -128 to 127
unsigned char     1       0 to 255
short             2       -32,768 to 32,767
unsigned short    2       0 to 65,535
int               4       -2,147,483,648 to 2,147,483,647
unsigned int      4       0 to 4,294,967,295
long              4       -2,147,483,648 to 2,147,483,647
unsigned long     4       0 to 4,294,967,295
float             4       3.4E+/-38 (7 digits)
double            8       1.7E+/-308 (15 digits)
long double      10       1.2E+/-4932 (19 digits)
```

The diligent student will notice that the only difference in these two lists are in the sizes and ranges of the **int** type variables, both signed and unsigned. The ANSI-C standard says that an **int** type has the "natural size suggested by the architecture of the execution environment", so the ranges for the compiler listed above matches the standard exactly.

One other point about the above table must be made at this time. The unadorned **char** is permitted to be either signed or unsigned at the discretion of the compiler writer. The writers of the Microsoft compiler chose to make **char** default to a **signed char**, as do most compiler writers, but you have a choice since most compilers provide a switch to select the default to **unsigned char**.

Some useful constants are available for your use in determining the range limits of the standard types. For example, the names INT_MIN and INT_MAX are available in the file "limits.h" as constants which can be used in your code. INT_MAX is the largest possible number that can be stored in an **int** type variable using the compiler that you are currently using. When you switch to a new compiler, which you will almost certainly do someday, INT_MAX will refer to the largest value that can be stored with that compiler. Even if you switch to a new operating system with 64 bits or even 128 bits, INT_MAX will still refer to the largest **int** available on your new system. The file "limits.h" contains a large number of such limits, all of which are available for your use simply by including the file in your program. It is a text file which can be opened in any editor and studied, a highly recommended exercise for you at this time.

LOTS OF VARIABLE TYPES

Example program ------> **LOTTYPES.C**

Load the file LOTTYPES.C and display it on your screen. This file contains most of the standard simple data types available in the programming language C. Consult your compiler documentaion for a complete list of all types avialable with your compiler. There are other types, but they are the compound types (ie - arrays and structures) that we will cover in due time in this tutorial.

Observe the file. First we define a simple **int**, followed by a **long int** . Next we have a **short int** which has a range that may be identical to that for the **int** variable. The **unsigned** is next and is defined as the same size as the **int** but with no sign. It should be pointed out that when the **long**, **short**, or **unsigned** is desired, the **int** is optional and is left out by most experienced programmers. Your compiler may differ significantly from the ranges given in the above table, so you should check the documentation for your compiler for the exact ranges for each type.

The **double** is a floating point number but covers a greater range than the **float** and has more significant digits for more precise calculations. It also requires more memory to store a value than the simple **float**. The **long double** will cover a much larger range and store more significant digits, but it will also take longer to do calculations because of the increased size of data being used.

Another diversion is in order at this point. Your compiler probably has no provision for floating point math, only **double** floating point math. It will promote a **float** to a **double** before doing calculations and therefore only one math library will be needed. Of course, this is transparent to you, so you don't need to worry about it. Because of this, you may think that it would be best to simply define every floating point variable as **double**, since they are promoted before use in any calculations, but that may not be a good idea. A **float** variable may require only 4 bytes of storage and a **double** may require 8 bytes of storage, so if you have a large volume of floating point data to store, the **double** will obviously require much more memory. If you don't need the additional range or significant digits, you should use the **float** type rather than the **double**. The compiler makes all floating point literals, such as the value 3.14159 in line 19, **double** constants by default. Some compilers will then issue a warning about line 19 because we are assigning a **double** to a **float.** You can safely ignore the warning at this time.

After defining the data types in the program under consideration, a numerical value is assigned to each of the defined variables in order to demonstrate the means of outputting each to the monitor.

SOME LATE ADDITIONS

As any programming language evolves, additional constructs are added to fill some previously overlooked need. Two new keywords have been added to C with the release of the ANSI-C standard. They are not illustrated in example programs, but they will be discussed here. The two new keywords are **const** and **volatile** and are used to tell the compiler that variables of these types will need special consideration. A constant is declared with the **const** keyword and declares a value that cannot be changed by the program. If you inadvertently try to modify an entity defined as a **const**, the compiler will generate an error. This is an indication to you that something is wrong. Declaring an entity as **const** allows the optimizer to do a better job which could make your program run a little faster. Since constants can never have a value assigned to them in the executable part of the program, they must always be initialized. If **volatile** is used, it declares a value that may be changed by the program but it may also be changed by some outside influence such as a clock update pulse incrementing the stored value. This prevents the optimizer from getting too ambitious and optimizing away something that it thinks will never be changed.

Examples of use in declaring constants of these two types are given as;

```
const int index1 = 2;
const index2 = 6;
const float big_value = 126.4;
```

```
volatile const int index3 = 12;
volatile int index4;
```

THE CONVERSION CHARACTERS

Following is a list of some of the conversion characters and the way they are used in the **printf()** statement. A complete list of all of the conversion characters should be included with the documentation for your compiler. You do not need to understand all of these at this time, but you should know that there is a lot of flexibility available when you are ready to use it.

```
d    decimal notation
i    decimal notation (new ANSI standard extension)
o    octal notation
x    hexadecimal notation
u    unsigned notation
c    character notation
s    string notation
f    floating point notation
```

Each of these is used following a percent sign to indicate the type of output conversion desired. The following fields may be added between those two characters.

```
-     left justification in its field
(n)   a number specifying minimum field width
.     to separate n from m
(m)   significant fractional digits for a float
l     to indicate a long
```

These are all used in the examples which are included in the program named LOTTYPES.C, with the exception of the string notation which will be covered later in this tutorial. Lines 33 through 35 illustrate how to set the field width to a desired width, and lines 39 and 40 illustrate how to set the field width under program control. The field width for the float type output in lines 43 through 47 should be self explanatory. Compile and run this program to see what effect the various fields have on the output.

You now have the ability to display any of the data fields in the previous programs and it would be to your advantage to go back and see if you can display some of the fields anyway you desire.

COMBINING THE VARIOUS TYPES

Example program ------> **COMBINE.C**

Examine the file named COMBINE.C for examples of combining variables of the various types in a program. Many times it is necessary to multiply an **int** type variable times a **float** type variable and C allows this by providing a strict set of rules it will follow in order to do such combinations.

Five variables of three different types are declared in lines 4 through 6, and three of them are initialized so we have some data to work with. Line 8 gives an example of adding an **int** variable to a **float** variable and assigning the result to a **char** type variable. The cast is used to control the type of addition and is indicated by defining the desired type within parentheses in front of the variable as shown. This forces each of the two variables to the **char** type prior to doing the addition. In some cases, when the cast is

used, the actual bit patterns must be modified internally in order to do the type coercion. Lines 9 through 11 perform the same addition by using different kinds of type casting to achieve the final result. Note that the addition is not the same in all three cases because the addition is done using different types, so could conceivably result in different answers.

Lines 13 through 15 illustrate the use of the cast to multiply two **float** variables. In two of the cases the intermediate results are cast to the **int** type, with the result being cast back to the **float** type. The observant student will notice that these three lines will not necessarily produce the same result.

Be sure to compile and execute this program. When you do, you may get a lot of type conversion warnings which can be ignored at this point. In this program, we are illustrating things that can be done with no regard to whether it is good to do so in a production program. Note that all of the warnings can be eliminated by including the proper cast when we use different types.

LOGICAL COMPARES

Example program ------> **COMPARES.C**

Load and view the file named COMPARES.C for many examples of compare statements in C. We begin by defining and initializing nine variables to use in the following compare statements.

The first group of compare statements represents the simplest kinds of compares because they simply compare two variables. Either variable could be replaced with a constant and still be a valid compare, but using two variables for the compare is the general case. The first compare checks to see if the value of **x** is equal to the value of **y** and it uses the double equal sign for the comparison. Since **x** is equal to **y**, the variable **z** will be assigned the value of -13. A single equal sign could be used here but it would have a different meaning as we will see shortly. The second comparison checks to see if the current value of **x** is greater than the current value of **z**.

The third compare introduces the not operator, the exclamation, which can be used to invert the result of any logical compare. The fourth checks for the value of **b** less than or equal to the value of **c**, and the last checks for the value of **r** not equal to the value of **s**. As we learned in the last chapter, if the result of the compare is true, the statement following the **if** clause will be executed and the results are given in the comments.

Note that "less than" and "greater than or equal to" are also available, but are not illustrated here.

It would be well to mention the different format used for the **if** statement in this example program. A carriage return is not required as a statement separator and by putting the conditional clause on the same line as the **if**, it adds to the readability of the overall program in this case.

MORE COMPARES

The compares in the second group are a bit more involved. Starting with the first compare, we find a rather strange looking set of conditions in the parentheses. To understand this we must understand just what a true or false is in the C language. A false is defined as a value of zero, and true is defined as any non-zero value. Any integer or character type of variable can be used for the result of a true/false test, or the result can be an implied integer or character.

Look at the first compare of the second group of compare statements. The conditional expression `"r !=`

s" will evaluate as a true since the value of **r** was set to 0.0 in line 13, so the result of the compare will be a non-zero value. With all ANSI-C compilers, it will be set to a 1. Good programming practice would be to not use the resulting 1 in any calculations, but only for logical control. Even though the two variables that are compared are **float** variables, the logical result will be of type **int**. There is no explicit variable to which it will be assigned so the result of the compare is an implied **int**. Finally, the resulting number, is assigned to the integer variable **x**. If double equal signs were used, the phantom value, namely 1, would be compared to the value of **x**, but since the single equal sign is used, the value 1 is simply assigned to the variable named **x**, as though the statement were not in parentheses. Finally, since the result of the assignment in the parentheses was non-zero, the entire expression is evaluated as true, and **z** is assigned the value of 1000. Thus we accomplished two things in this statement, we assigned **x** a new value, and we assigned **z** the value of 1000. We covered a lot in this statement so you may wish to review it before going on. The important things to remember are the values that define true and false, and the fact that several things can be assigned in a conditional statement. The value assigned to the variable **x** was probably a 1, but remember that the only requirement is that it is nonzero. The ANSI-C standard says that the result of a comparison operation, (>, >=, <, or <=) must be 1 or 0, but does not state the result of an equality operation. If you assume 0 or 1 will be returned, and only use it for control, you will not get into trouble.

The example in line 20 should help clear up some of the above in your mind. In this example, **x** is assigned the value of **y**, and since the result is 11, the condition is non-zero, which is true, and the variable **z** is assigned 222.

The third example of the second group in line 21, compares the value of **x** to zero. If the result is true, meaning that if **x** is not zero, then **z** is assigned the value of 333, which it will be. The last example in this group illustrates the same concept, since the result will be true if **x** is non-zero. The compare to zero in line 21 is not actually needed and the result of the compare is true. The third and fourth examples of this group are therefore logically identical. Of course we assign a different value to **z** in each case.

ADDITIONAL COMPARE CONCEPTS

The third group of compares will introduce some additional concepts, namely the logical "and" and the logical "or" operators. We assign the value of 77 to the three integer variables simply to get started again with some defined values. The first compare of the third group contains the new control &&, which is the logical "and" which results in a true if both sides of the "and" are true. The entire statement reads, if **x** equals **y** and if **x** equals 77 then the result is true. Since this is true, the variable **z** is set equal to 33. Note that only integral types can be "anded", so **float** and **double** types cannot be used here.

The next compare in this group introduces the || operator which is the logical "or" operator which results in a true if either side of the "or" is true. The statement reads, if **x** is greater than **y** or if **z** is greater than 12 then the result is true. Since **z** is greater than 12, it doesn't matter if **x** is greater than **y**, because only one of the two conditions must be true for the result to be true. The result is true, therefore **z** will be assigned the value of 22. Once again, **float** and **double** cannot be "ored".

LOGICAL EVALUATION (SHORT CIRCUIT)

When a compound expression is evaluated, the evaluation proceeds from left to right and as soon as the result of the outcome is assured, evaluation stops. Therefore, in the case of an "and" evaluation, when one of the terms evaluates to false, evaluation is discontinued because additional true terms cannot make

the result ever become true. In the case of an "or" evaluation, if any of the terms is found to be true, evaluation stops because it will be impossible for additional terms to cause the result to be false. In the case of additionally nested terms, the above rules will be applied to each of the nested levels. This is called short-circuit evaluation since the remaining terms are not evaluated.

Going on to the next example in group three in line 29, we find three simple variables used in the conditional part of the compare. Since all three are non-zero, all three are true, and therefore the "and" of the three variables is true, leading to the result being true, and **z** is assigned the value of 11. Note that since the variables, **r**, **s**, and **t** are **float** type variables, they could not be used this way.

Continuing on to line 30 we find three assignment statements in the compare part of the **if** statement. If you understood the above discussion, you should have no difficulty understanding that the three variables are assigned their respective new values, and the result of all three are non-zero, leading to a resulting value of true.

THIS IS A TRICK, BE CAREFUL

The last example of the third group contains a bit of a trick, but since we have covered it above, it is nothing new to you. Notice that the first part of the compare evaluates to false since **x** is not currently 2. The remaining parts of the compare are not evaluated, because it is a logical "and" so it will definitely be resolved as a false because the first term is false. If the program was dependent on the value of **y** being set to 3 in the next part of the compare, it will fail because evaluation will cease following the false found in the first term. Likewise, the variable named **z** will not be set to 4, and the variable **r** will not be changed. This is because C uses short circuit evaluation as discussed earlier.

POTENTIAL PROBLEM AREAS

The last group of compares illustrate three possibilities for getting into a bit of trouble. All three have the common result that the variable **z** will not be handled properly, but for different reasons. In line 37, the compare evaluates as true, but the semicolon following the second parentheses terminates the **if** clause, and the assignment statement involving **z** is always executed as the next statement. The **if** therefore has no effect because of the misplaced semicolon. This is actually a null statement and is legal in C, but the programmer probably did not intend to include the extra semicolon.

The statement in line 38 is much more straightforward because the variable **x** will always be equal to itself, therefore the inequality will never be true, and the entire statement will never do a thing, but is wasted effort. The statement in line 39 will always assign 0 to **x** and the compare will therefore always be false, never executing the conditional part of the **if** statement.

The conditional statement is extremely important and must be thoroughly understood to write efficient C programs. If any part of this discussion is unclear in your mind, restudy it until you are confident that you understand it thoroughly before proceeding onward. Compile and run this program. You may gets lots of conversion warnings which you can either ignore or fix up the code with casts to eliminate. Add some printout to see the results of some of the operations.

THE CRYPTIC PART OF C

Example program ------> **CRYPTIC.C**

There are three constructs used in C that make no sense at all when first encountered because they are not

intuitive, but they may increase the efficiency of the compiled code and are used extensively by experienced C programmers. You should therefore be exposed to them and learn to use them because they will appear in most, if not all, of the programs you see in the publications. Load and examine the file named CRYPTIC.C for examples of the three new constructs.

In this program, some variables are defined and initialized in the same statements for use later. The statement in line 8 simply adds 1 to the value of **x**, and should come as no surprise to you. The next two statements also add one to the value of **x**, but it is not intuitive that this is what happens. It is simply by definition that this is true. Therefore, by definition of the C language, a double plus sign either before or after a variable increments that variable by 1. Additionally, if the plus signs are before the variable, the variable is incremented before it is used, and if the plus signs are after the variable, the variable is used, then incremented. In line 11, the value of **y** is assigned to the variable **z**, then **y** is incremented because the plus signs are after the variable **y**. In the last statement of the incrementing group of example statements, line 12, the value of **y** is incremented then its value is assigned to the variable **z**. To use the proper terminology, line 9 uses the postincrement operator and line 10 uses the preincrement operator.

The next group of statements illustrate decrementing a variable by one. The definition works exactly the same way for decrementing as it does for incrementing. If the minus signs are before the variable, the variable is decremented, then used, and if the minus signs are after the variable, the variable is used, then decremented. The proper terminology is the postdecrement operator and the predecrement operator.

You will use this construct a lot in your C programs.

THE CRYPTIC ARITHMETIC OPERATOR

Another useful but cryptic operator is the arithmetic operator. This operator is used to modify any variable by some constant value. The statement in line 23 adds 12 to the value of the variable **a**. The statement in line 24 does the same, but once again, it is not intuitive that they are the same. Any of the four basic functions of arithmetic, +, -, *, or /, can be handled in this way, by putting the operation desired in front of the equal sign and eliminating the second reference to the variable name. It should be noted that the expression on the right side of the arithmetic operator can be any valid expression, the examples are kept simple for your introduction to this new operator.

Just like the incrementing and decrementing operators, the arithmetic operator is used extensively by experienced C programmers and it would pay you well to understand it thoroughly.

THE CONDITIONAL EXPRESSION

The conditional expression is just as cryptic as the last two, but once again it is very useful so it would pay you to understand it. It consists of three expressions separated by a question mark and a colon. The expression prior to the question mark is evaluated to determine if it is true or false. If it is true, the expression between the question mark and the colon is evaluated, and if the compare expression is not true, the expression following the colon is evaluated. The result of one of the evaluations is used for the assignment as illustrated in line 30. The final result is identical to that of an **if** statement with an **else** clause. This is illustrated by the example in lines 32 through 35 of this group of statements. The conditional expression has the advantage of more compact code that may compile to fewer machine instructions in the final program.

Lines 37 and 38 of this example program are given to illustrate a very compact way to assign the greater

of the two variables **a** or **b** to the variable **c**, and to assign the lessor of the same two variables to the variable **c**. Notice how efficient the code is in these two examples.

TO BE CRYPTIC OR NOT TO BE CRYPTIC

Several students of C have stated that they didn't like these three cryptic constructs and that they would simply never use them. This would be fine if they never have to read anybody else's program, or use any other programs within their own. You will find many functions that you wish to use within a program but need a small modification to use it, requiring you to understand another person's code. It would therefore be to your advantage to learn these new constructs, and use them. They will be used in the remainder of this tutorial, so you will be exposed to them.

This has been a long chapter but it contained important material to get you started in using C. In the next chapter, we will go on to the building blocks of C, the functions. At that point, you will have enough of the basic materials to allow you to begin writing meaningful programs.

STYLE ISSUES

We have no specific issues of style in this chapter other than some of the coding styles illustrated in the example programs. Most of these programs are very nontypical of real C programs because there is never a need to list all of the possible compares in a real program, for example. You can use the example programs as a guide to good style even though they are not real programs.

WHAT IS AN l-value AND AN r-value?

You will sometimes see a reference to an l-value or a r-value in writings about C or in the documentation for your C compiler. Every variable has an r-value which is defined as the actual value stored in the variable, and it also has an l-value which is defined as the name of the variable. Therefore, the variable depicted graphically in figure 4-1 has an l-value of **index**, and an r-value of 137 since 137 is the value stored in the variable at this time.

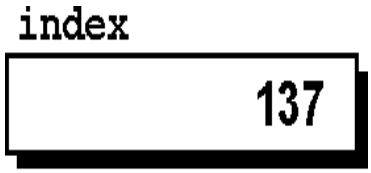

Figure 4-1

The definition for this variable would be given as follows;

```
int index = 137;
```

PROGRAMMING EXERCISES

1. Write a program that will count from 1 to 12 and print the count, and its square, for each count.
2. Write a program that counts from 1 to 12 and prints the count and its inversion to 5 decimal places for each count. This will require a floating point number.
3. Write a program that will count from 1 to 100 and print only those values between 32 and 39, one to a line. Use the incrementing operator for this program.

Return to Table of Contents

Advance to Chapter 5

Gordon Dodrill - dodrill@swcp.com - Please email any comments or suggestions.

C Tutorial - Chapter 5

FUNCTIONS, VARIABLES, AND PROTOTYPES

OUR FIRST USER DEFINED FUNCTION

Example program ------> **SUMSQRES.C**

Load and examine the file SUMSQRES.C for an example of a C program with functions. Actually this is not the first function we have encountered because the main program we have been using all along is technically a function, as is the **printf()** function. The **printf()** function is a library function that was supplied with your compiler.

We will finally define what line 3 is for in this chapter but not until we get to about the fourth example program, so continue to wait patiently and ignore that line for the time being.

Notice the executable part of this program which begins in line 10 with a line of code that simply says "`header();`", which is the way to call any function. The parentheses are required because the C compiler uses them to determine that it is a function call and not simply a misplaced variable. When the program comes to this line of code, the function named **header()** is called, its statements are executed, and control returns to the statement following this call. Continuing on, we come to a **for** loop which will be executed 7 times and which calls another function named **square()** each time through the loop. Finally, a function named **ending()** will be called and executed. For the moment ignore the variable name **index** in the parentheses of the call to **square()**. We have seen that this program calls a header, 7 square calls, and an ending. Now we need to define the functions.

DEFINING THE FUNCTIONS

Following the main program you will see a function beginning in line 19 that follows all of the rules set forth so far for a main program except that it is named **header()**. This is the function which is called from line 10 of the main program. Each of these statements are executed, and when they are all complete, control returns to the main program, or more properly, the **main()** function.

The first statement sets the variable named **sum** equal to zero because we plan to use it to accumulate a sum of squares. Since the variable named **sum** is defined prior to the main program, it is available for use in any of the functions which are defined after the variable is defined. It is called a global variable, and its scope is the entire program including all functions. It is also sometimes referred to as a file variable because it is available throughout the file. More will be said about the scope of variables near the end of this chapter. The statement in line 22 outputs a header message to the monitor. Program control then returns to the **main()** function since there are no additional statements to execute in this function. Essentially, we drop out of the bottom of the function and return to the caller where we begin executing statements immediately following where the call was made from.

It should be clear to you that the two executable lines from this function could be moved to the main program, replacing the header call, and the program would do exactly the same thing that it does as it is now written. This does not minimize the value of functions, it merely illustrates the operation of this simple function in a simple way. You will find functions to be very valuable in C programming.

PASSING A VALUE TO A FUNCTION (CLASSIC METHOD)

Going back to the main program, and the **for** loop specifically, we find the new construct from the end of the last lesson used in the last part of the **for** loop, namely the `index++` used in line 11. You should get familiar with this construct, as you will see it in a lot of C programs.

In the call to the function named **square()**, we have an added feature, the variable name **index** within the parentheses. This is an indication to the compiler that when you jump to the function, you wish to take along the value of **index** to use during the execution of that function. Looking ahead at the function named **square()** in line 26, we find that another variable name is enclosed in its parentheses, the variable **number**. This is the name we prefer to call the variable passed to the function when we are executing code within the function. We can call it anything we wish as long as it follows the rules of naming an identifier and is not a keyword. Since the function must know what type the variable is, it is defined following the function name but before the opening brace of the function itself. Therefore, line 23 containing the expression "`int number;`" tells the function that the value passed to it will be an **int** type variable. With all of that out of the way, we now have the value of **index** from the main program passed to the function **square()**, but renamed **number**, and available for use within the function. This is the classic style of defining function variables and has been in use since C was originally defined. A newer and much better method is gaining in popularity due to its many benefits and will be discussed later in this chapter.

Following the opening brace of the function, we define another variable named **numsq** for use only within the function itself, (more about that later) and proceed with the required calculations. We set the variable named **numsq** equal to the square of the value of **number**, then add **numsq** to the current total stored in the variable named **sum**. You should remember that the expression "`sum += numsq;`" has the same meaning as "`sum = sum + numsq;`" from the last lesson. We print the number and its square in line 33, and return to the main program.

MORE ABOUT PASSING A VALUE TO A FUNCTION

When we passed the value of the variable named **index** to the function, a little more happened than meets the eye. We didn't pass the variable named **index** to the function, we actually passed a copy of the value. In this way, the original value is protected from accidental corruption by the called function. We could have modified the variable named **number** in any way we wished in the function named **square(),** and when we returned to the main program, the variable named **index** would not have been modified. We thus protect the value of a variable in the calling function from being accidentally corrupted, but we cannot return a value to the calling function from a called function using this technique. We will find a well defined method of returning values to the **main()** function or to any calling function when we get to arrays and another method when we get to pointers. Until then, the only way you will be able to communicate back to the calling function will be with global variables. We have already hinted at global variables above, and will discuss them in detail later in this chapter.

Continuing in the **main()** function, we come to the last function call, the call to the function named **ending()** in line 13. This line calls the last function which has no local variables defined. It prints out a message with the value of the variable **sum** contained in it to end the program. The program ends by returning to the **main()** functoin and finding nothing else to do, so the program terminates. Compile and run this program and observe the output.

NOW TO CONFESS A LITTLE LIE

I told you a short time ago that the only way to get a value back to the calling function was through use of a global variable, but there is another way which we will discuss after you load and display the program named SQUARES.C. In this example program we will see that it is simple to return a single value from a called function to the calling function. But once again, it is true that to return more than one value, we will need to study either arrays or pointers.

Example program ------> **SQUARES.C**

In the **main()** function, we define two integers and begin a **for** loop in line 8 which will be executed 8 times. The first statement within the **for** loop is "`y = squ(x);`", which is a new and rather strange looking construct. From past experience, we should have no trouble understanding that the **squ(x)** portion of the statement is a call to the function named **squ()** taking along the value of **x** as a parameter. Looking ahead to line 20 of the function itself, we find that the function prefers to call the input variable **input**, and it proceeds to square the value of **input** and call the result **square**. Finally, a new kind of a statement appears in line 26, the **return** statement. The value within the parentheses is assigned to the function itself and is returned as a usable value in the main program. Thus, the function call "`squ(x)`" is assigned the value of the square and returned to the main program such that the variable named **y** is then set equal to that value. If the variable named **x** were therefore assigned the value 4 prior to this call, **y** would then be set to 16 as a result of the code in line 10.

The parentheses around the return value in line 26 are not required, but are included by many experienced C programmers.

Another way to think of this is to consider the grouping of characters **squ(x)** as another variable with a value that is the square of **x**, and this new variable can be used any place it is legal to use a variable of its type. The values of the variables **x** and **y** are then printed out.

To illustrate that the grouping of **squ(x)** can be thought of as just another variable, another **for** loop is introduced in line 14 in which the function call is placed within the **printf()** statement rather than assigning it to a new variable.

One last point must be made, the type of variable returned must be defined in order to make sense of the data, but the compiler will default the type to **int** if none is specified. If any other type is desired, it must be explicitly defined. How to do this will be demonstrated in the next example program. We are simply using the default return type in this program.

Be sure to compile and run this program which also uses the classic method of defining function variables. Once again, any warnings can be ignored.

FLOATING POINT FUNCTIONS

Example program ------> **FLOATSQ.C**

Load the program FLOATSQ.C for an example of a function in the classic style with a **float** type of return. It begins by defining a global floating point variable named **z** which we will use later. Then in the main part of the program, an integer is defined, followed by two floating point variables, and then by two strange looking definitions. The expressions **sqr()** and **glsqr()** look like function calls. This is the proper way to define that a function will return a value that is not of type **int**, but of some other type, in this case

float. This tells the compiler that when a value is returned from either of these two functions, it will be of type **float**. This is, once again, the classic method of defining functions and is all but obsolete now. Note that neither function is actually called by the code in line 9, these only declare the return type for these two functions.

Now refer to the function named **sqr()** starting in line 29 and you will see that the function name is preceded by the keyword **float**. This is an indication to the compiler that this function will return a value of type **float** to any program that calls it. The type of the function return is now compatible with the call to it. The line following the function name contains `float inval;`, which indicates to the compiler that the variable passed to this function from the calling program will be of type **float**. Since we told the caller we would be returning a **float** type, we actually return the value in line 35 so everything matches up.

The function named **glsqr()** beginning in line 39, will also return a **float** type variable, but it uses a global variable for input. It does the squaring right within the **return** statement and therefore has no need to define a separate variable to store the product. The function **sqr()** could have done the squaring right in the return also, but was done separately as an illustration of what can be done.

The overall structure of this program should pose no problem and will not be discussed in any further detail. As is customary with all example programs, compile and run this program and ignore any warnings you may get.

THE CLASSIC STYLE

The three programs we have studied in this chapter so far use the classic style of function definition. Although this was the first style defined for C, it is rapidly being replaced with a more modern method of function definition because the modern method does so much for you in detecting and flagging errors. As you read articles on C, you will see programs written in the classic style, so you need to be capable of reading them. This is the reason the classic style was included in this chapter. It would be highly recommended, however, that you learn and use the modern method which will be covered shortly in this tutorial. In fact, you are advised to never use the classic style for any of your programming efforts.

The book by Kernigan and Ritchie, "The C Programming Language - Second Edition" is the definitive text of the classic style of C programming.

The remainder of this tutorial will use the modern method as recommended and defined by the ANSI-C standard. If you have an older compiler, it may not work on some of these files and it will be up to you to modify the programs as needed to conform to the classic style. Actually, the ANSI-C standard is used so universally, if you have a non-ANSI compiler you should use it only as a doorstop and purchase a good ANSI compatible compiler for the rest of your studies.

THE RETURN TYPE OF main()

In the original K&R definition of C, all functions returned an **int** type variable by default, unless the author specified something different. Since explicitly returning a value when leaving a function was optional, most C was written in the following manner;

```
main()
{
   ...
```

```
}
```

When prototyping was added to the language (which we will study shortly), many programmers apparently thought the **main()** function didn't return anything, so used the **void** type for a return and it became a common practice to write the **main()** function as follows;

```
void main()
{
   ...
}
```

When the ANSI-C standard was finalized the only return type approved by the standard is an **int** type variable. A good compiler will check that the program actually returns a value by requiring that an integer is returned from each exit point. This led to the following form for **main()**;

```
int main()
{
   ...
   return 0;
}
```

Apparently because of the inertia behind the use of a **void** return, many compiler writers added the **void** return as an extension to permit the use of legacy code without modifications. Some compilers therefore support the **void** return but the **int** return is the only method approved by the ANSI-C standard.

In order to make your code as portable as possible, you should always use the last form above.

You can finally see why we have been adding the line that returns a value of zero to the operating system. This indicates to the operating system that the program executed normally.

SCOPE OF VARIABLES

Example program ------> **SCOPE.C**

Load the next program, SCOPE.C, and display it for a discussion of the scope of variables in a program. You can ignore the 4 statements in lines 2 through 5 of this program for a few moments. We will discuss them later. We will spend a good deal of time in this program and cover a lot of new topics. Many of the topics covered here will not seem to be particularly useful, but stay with it because they are very important.

WHAT IS A GLOBAL VARIABLE?

The variable defined in line 7 is a global variable named **count** which is available to any function in the program since it is defined before any of the functions. It is always available because it exists during all the time that the program is being executed. (That will make sense shortly.) Farther down in the program, another global variable named **counter** is defined in line 29 which is also global but is not available to the **main()** function since it is defined following the **main()** function. A global variable is any variable that is defined outside of any function. Note that both of these variables are sometimes referred to as external variables because they are external to any functions, and they are sometimes also called file variables.

Global variables are automatically initialized to zero when they are defined. Therefore, the variables named **count** and **counter** will both be initialized to a value of 0.

Return to the **main()** function and you will see the variable named **index** defined as an **int** in line 11. Ignore the word **register** for the moment. This variable is only available within the **main()** function because that is where it is defined. In addition, it is an automatic variable, which means that it only comes into existence when the function in which it is contained is invoked, and ceases to exist when the function is finished. This really means nothing here because the **main()** function is always in operation, even when it gives control to another function. Another integer is defined within the **for** loop braces named **stuff**. Any pairing of braces can contain variable definitions which will be valid and available only while the program is executing statements within those braces. The variables will be automatic variables and will cease to exist when execution leaves the braces. The variable named **stuff** will therefore be created and destroyed 8 times, once for each pass through the loop.

MORE ON AUTOMATIC VARIABLES

Observe the function named **head1()** in line 30 which looks a little funny because of **void** being used twice. The purpose of the use of the word **void** will be explained shortly. The function contains a variable named **index**, which has nothing to do with the variable named **index** in line 11 of the **main()** function, except that both are automatic variables. When the program is not actually executing statements in this function, this variable named **index** does not even exist. When **head1()** is called, the variable is generated, and when **head1()** completes its task, the variable in **head1()** named **index** is eliminated completely from existence. (The automatic variable is stored on the stack. This topic will be covered later.) Keep in mind however that this does not affect the variable of the same name in the **main()** function, since it is a completely separate entity.

Automatic variables therefore, are automatically generated and disposed of when needed. The important thing to remember is that from one call of a function to the next call, the value of an automatic variable is not preserved and must therefore be reinitialized.

WHAT ARE STATIC VARIABLES?

An additional variable type must be mentioned at this point, the static variable. By putting the keyword **static** in front of a variable definition within a function, the variable or variables in that definition are static variables and will stay in existence from call to call of the particular function. A static variable is initialized once, at load time, and is never reinitialized during execution of the program.

By putting the **static** keyword in front of an external variable, one outside of any function, it makes the variable private and not accessible to use in any other file. (This is a completely different use of the same keyword.) This implies that it is possible to refer to external variables in other separately compiled files, and that is true. Examples of this usage will be given in chapter 14 of this tutorial. They are not illustrated here.

USING THE SAME NAME AGAIN

Refer to the function named **head2()**. It contains another definition of the variable named **count**. Even though **count** has already been defined as a global variable in line 7, it is perfectly all right to reuse the name in this function. It is a completely new variable that has nothing to do with the global variable of the same name, and causes the global variable to be unavailable within this function. This allows you to

write programs using existing functions without worrying about what names were used for global variables or in other functions because there can be no conflict. You only need to worry about the variables that interface with the functions.

WHAT IS A REGISTER VARIABLE?

Now to fulfill a promise made earlier about what a **register** variable is. A computer can keep data in a register or in memory. A register is much faster in operation than memory but there are very few registers available for the programmer to use. If there are certain variables that are used extensively in a program, you can designate that those variables are to be stored in a register in order to speed up the execution of the program. The method of doing this is illustrated in line 11. Your compiler probably allows you to use one or more register variables and will ignore additional requests if you request more than are available. The documentation for your compiler should list how many registers are available with your compiler. It will also inform you of what types of variables can be stored in a register. If your compiler does not allow the use of register variables, the register request will simply be ignored.

WHAT IS PROTOTYPING?

A prototype is a model of a real thing and when programming in ANSI-C, you have the ability to define a model of each function for the compiler. The compiler can then use the model to check each of your calls to the function and determine if you have used the correct number of arguments in the function call and if they are of the correct type. By using prototypes, you let the compiler do some additional error checking for you. The ANSI standard for C contains prototyping as part of its recommended standard. Every ANSI-C compiler will have prototyping available, so you should learn to use it. Much more will be said about prototyping throughout the remainder of this tutorial.

Returning to lines 3, 4, and 5 in SCOPE.C, we have the prototypes for the three functions contained within the program. The first **void** in each line tells the compiler that these particular functions do not return a value, so that the compiler would flag the statement `index = head1();` as an error because nothing is returned to assign to the variable named **index**. The word **void** within the parentheses tells the compiler that this function requires no parameters and if a variable were included, it would be an error and the compiler would issue a warning message. If you wrote the statement `head1(index);`, it would be a error. This allows you to use type checking when programming in C in much the same manner that it is used in Pascal, Modula-2, or Ada, although the type checking in C is relatively weak.

You should begin using prototype checking at this time, for all of the functions you define. Your compiler may have an option that will require a prototype for every function. This should be enabled and left enabled. Check your documentation for the details of how to do it. Prototyping will be used throughout the remainder of this tutorial. If your compiler does not support prototyping and the modern method of function definition, you will have to modify the remaining example programs. A much better solution would be to purchase a better compiler.

Line 2 of SCOPE.C tells the system to go to the standard directory where include files are stored and get a copy of the file named stdio.h which contains the prototypes for the standard input and output functions so they can be checked for proper variable types. Don't worry about the include yet, it will be covered in detail later in this tutorial. Be sure to compile and execute this program.

STANDARD FUNCTION LIBRARIES

Every compiler comes with some standard predefined functions which are available for your use. These are mostly input/output functions, character and string manipulation functions, and math functions. We will cover many of these in subsequent chapters. Prototypes are defined for you by the writer of your compiler for all of the functions that are included with your compiler. A few minutes spent studying your reference guide will give you an insight in where the prototypes are defined for each of the functions. Most compilers have additional functions predefined that are not standard but allow the programmer to get the most out of his particular computer. In the case of the IBM-PC and compatibles, most of these functions allow the programmer to use the BIOS services available in the operating system, or to write directly to the video monitor or to any place in memory. These will not be covered in any detail as you will be able to study these unique aspects of your compiler on your own. Several of these kinds of functions are used in the example programs in chapter 14.

WHAT IS RECURSION?

Example program ------> **RECURSON.C**

Recursion is another of those programming techniques that seem very intimidating the first time you come across it, but if you will load and display the example program named RECURSON.C, we will take all of the mystery out of it. This is probably the simplest recursive program that it is possible to write and it is therefore a stupid program in actual practice, but for purposes of illustration, it is excellent.

Recursion is nothing more than a function that calls itself. It is therefore in a loop which must have a way of terminating. In the program on your monitor, the variable named **index** is set to 8 in line 9, and is used as the argument to the function named **count_dn()**. The function simply decrements the variable, prints it out in a message, and if the variable is greater than zero, it calls itself, where it decrements the variable again, prints it, etc. etc. etc. Finally, the variable will reach zero, and the function will not call itself again. Instead, it will return to the prior time it called itself, and return again, and again, until finally it will return to the **main()** function and from there return to the operating system.

For purposes of understanding you can think of it as having 8 copies of the function named **count_dn()** available and it simply called all of them one at a time, keeping track of which copy it was in at any given time. That is not what actually happened, but it is a reasonable illustration for you to begin understanding what it was really doing.

WHAT DID IT DO?

A better explanation of what actually happened is in order. When you called the function from itself, it stored all of the variables and all of the internal flags it needs to complete the function in a block somewhere. The next time it called itself, it did the same thing, creating and storing another block of everything it needed to complete that function call. It continued making these blocks and storing them away until it reached the last function when it started retrieving the blocks of data, and using them to complete each function call. The blocks were stored on an internal part of the computer called the stack. This is a part of memory carefully organized to store data just as described above. It is beyond the scope of this tutorial to describe the stack in detail, but it would be good for your programming experience to read some material describing the stack. A stack is used in nearly all modern computers for internal housekeeping chores.

In using recursion, you may desire to write a program with indirect recursion as opposed to the direct recursion described above. Indirect recursion would be when a function A calls the function B, which in

turn calls A, etc. This is entirely permissible, the system will take care of putting the necessary things on the stack and retrieving them when needed again. There is no reason why you could not have three functions calling each other in a circle, or four, or five, etc. The C compiler will take care of all of the details for you.

The thing you must remember about recursion is that at some point, something must go to zero, or reach some predefined point to terminate the loop. If not, you will have an infinite loop, and the stack will fill up and overflow, giving you an error and stopping the program rather abruptly.

ANOTHER EXAMPLE OF RECURSION

Example program ------> **BACKWARD.C**

The program named BACKWARD.C is another example of recursion, so load it and display it on your screen at this time. This program is similar to the last one except that it uses a character array. Each successive call to the function named **forward_and_backwards()** causes one character of the message to be printed. Additionally, each time the function ends, one of the characters is printed again, this time backwards as the string of recursive function calls is retraced.

This program uses the modern method of function definition and includes full prototype definitions. The modern method of function definition moves the types of the variables into the parentheses along with the variable names themselves. The final result is that the line containing the function definition looks more like the corresponding line in a language with relatively strong type checking such as Pascal, Modula-2, or Ada. The prototype in line 5 is simply a copy of the function header in line 20 followed by a semicolon. The designers of C even allow you to include a variable name along with each type. The name is ignored by the compiler but including the name in the prototype could give you a good idea of how the variable is used, acting like a comment.

Don't worry about the character array defined in line 9 or the other new material presented here. After you complete chapter 7 of this tutorial, this program will make sense. It was felt that introducing a second example of recursion was important so this file is included here. You will note that this program actually does something useful with recursion, but it would be mighty easy to duplicate the action of the program without recursion. We will study some programs later where recursion is required.

Compile and run this program and observe the results.

THE FLOAT SQUARE PROGRAM WITH PROTOTYPES

Example program ------> **FLOATSQ2.C**

Load and display the program named FLOATSQ2.C which is an exact copy of the program FLOATSQ.C which we considered earlier with prototyping added. The use of prototyping is a good practice for all C programmers to get into.

Several things should be mentioned about this program. First, the word **float** at the beginning of lines 32 and 41 indicate to the compiler that these functions are functions that return **float** type values. Also, since prototypes for the functions are given before **main()**, the functions are not required to be identified in line 12 as they were in line 9 of FLOATSQ.C earlier in this chapter. Notice also that the type of the variable named **inval** is included within the parentheses in lines 4 and 32.

After you compile and execute this program, ignoring any warnings, remove the parameter from line 17 to see what kind of error message you get.

A CONFUSING PROBLEM THAT COMES UP AT TIMES

Suppose we wrote the following line of code in the FLOATSQ2.C program,

```
printf(" ... ", sqr(5.0), glsqr());
```

It may come as a surprise to you, but the order in which the two functions are called, is undefined as far as the ANSI-C standard is concerned. One compiler writer may call **sqr()** first, and another may call **glsqr()** first, and either method is correct. In this case, it makes no difference which is called first, but in some cases it does matter such as if something is printed out in both functions. The result returned from each call will be used in the correct location, but the order of evaluation is undefined. Rest assured that this will come up at some point in your future programming efforts so you need to be aware of it.

MORE STYLE ISSUES

Example program ------> **STYLE2.C**

The example named STYLE2.C is given as an illustration of various ways to format a function. You will note different ways to define the input parameters. Examples three and four are both the same style, but example four illustrates the style when nothing is passed in or returned. This style states very clearly that nothing is needed or returned and it cannot be construed as an oversight. Spend some time studying these function examples, then begin developing the style you will use. If you are like most programmers, you will develop a style that you plan to use forever, then change it every few months or on every new project.

PROGRAMMING EXERCISES

1. Rewrite TEMPCONV.C from an earlier chapter, and move the temperature calculation to a function.
2. Write a program that writes your name on the monitor 10 times by calling a function to do the writing. Move the called function ahead of the main function to see if your C compiler will allow it.
3. Add prototyping to the programs named SUMSQRES.C and SQUARES.C, and change the function definitions to the modern method.

Return to Table of Contents

Advance to Chapter 6

Gordon Dodrill - dodrill@swcp.com - Please email any comments or suggestions.

C Tutorial - Chapter 6

THE C PREPROCESSOR

AIDS TO CLEAR PROGRAMMING

The preprocessor is a program that is executed just prior to the execution of the compiler. It's operation is transparent to you but it does a very important job. It removes all comments from the source and performs a lot of textual substitution based on your code, passing the result to the compiler for the actual compilation of your code.

Example program ------> **DEFINE.C**

Load and display the file named DEFINE.C for your first look at some defines and macros. Notice lines 4 through 7 of the program, each starting with **#define.** This is the way all defines and macros are declared. Before the actual compilation starts, the compiler goes through a preprocessor pass to resolve all of the defines. In the present case, it will find every place in the program where the word START is found and it will replace it with the 0 since that is the definition. The compiler itself will never see the word START, so as far as the compiler is concerned, the zeros were always there. Note that if the word is found in a string constant or in a comment, it will not be changed.

It should be clear to you that putting the word START in your program instead of the numeral 0 is only a convenience to you and actually acts like a comment since the word START helps you to understand what the zero is used for.

In the case of a very small program, such as that before you, it doesn't really matter what you use. If, however, you had a 2000 line program before you with 27 references to START, it would be a completely different matter. If you wanted to change all of the STARTs in the program to a new number, it would be simple to change the one **#define** statement to the new value. If this technique were not used, it would be difficult to find and change all of the references to it manually, and possibly disastrous if you missed one or two of the references.

In the same manner, the preprocessor will find all occurrences of the word ENDING and change them to 9, then the compiler will operate on the changed file with no knowledge that ENDING ever existed.

It is a fairly common practice in C programming to use all capital letters for a symbolic constant such as START and ENDING and use all lower case letters for variable names. You can use any method you choose since it is mostly a matter of personal taste.

WHAT IS A MACRO?

A macro is nothing more than another define, but since it is capable of at least appearing to perform some logical decisions or some math functions, it has a unique name. Consider line 6 of the program on your monitor for an example of a macro. In this case, anytime the preprocessor finds the word MAX followed by a group in parentheses, it expects to find two terms in the parentheses and will do a replacement of the terms into the second part of the definition. Thus the first term will replace every **A** in the second part of the definition and the second term will replace every **B** in the second part of the definition. When line 16 of the program is reached, **index** will be substituted for every **A**, and **count** will be substituted for every

B. Therefore, before line 16 is given to the compiler, it will be modified to the following;

```
mx = ((index)>(count)?(index):(count))
```

Once again, it must be stated that string constants and comments will not be affected. Remembering the cryptic construct we studied a couple of chapters ago will reveal that **mx** will receive the maximum value of **index** or **count**. In like manner, the MIN macro will result in **mn** receiving the minimum value of **index** or **count**. These two particular macros are very common in C programs.

When defining a macro, it is imperative that there is no space between the macro name and the opening parenthesis. If there is a space, the compiler cannot determine that it is a macro, but will handle it like a simple substitution define statement.

The results of the macro usage are then printed out in line 18. There are a lot of seemingly extra parentheses in the macro definition but they are not extra, they are essential. We will discuss the extra parentheses in our next example program. Be sure to compile and execute DEFINE.C before going on to the next example program.

LET'S LOOK AT A WRONG MACRO

Example program ------> **MACRO.C**

Load the file named MACRO.C and display it on your screen for a better look at a macro and its use. Line 4 defines a macro named WRONG that appears to evaluate the cube of **A**, and indeed it does in some cases, but it fails miserably in others. The second macro named CUBE actually does get the cube in most but not all cases. We will soon see why it fails in some cases.

Consider the program itself where the CUBE of **i+offset** is calculated in line 20. If **i** is 1, which it is the first time through, then we will be looking for the cube of 1+5 = 6, which will result in 216. When using CUBE, we group the values like this, (1+5)*(1+5)*(1+5) = 6*6*6 = 216. However, when we use WRONG, we group them as 1+5*1+5*1+5 = 1+5+5+5 = 16 which is a wrong answer. The parentheses are therefore required to properly group the variables together. It should be clear to you that either CUBE or WRONG would arrive at a correct answer for a single term replacement such as we did in the last program. The correct values of the cube and the square of the numbers are printed out as well as the wrong values for your inspection.

In line 7 we define the macro ADD_WRONG according to the above rules but we still have a problem when we try to use the macro in lines 28 and 29. In line 29 when we say we want the program to calculate 5*ADD_WRONG(i) with i = 1, we get the result 5*1 + 1 which evaluates to 5 + 1 or 6, and this is most assuredly not what we had in mind. We really wanted the result to be 5*(1 + 1) = 5*2 = 10 which is the answer we get when we use the macro named ADD_RIGHT, because of the extra parentheses around the entire expression in the definition given in line 8. A little time spent studying the program and the result will be worth your effort in understanding how to use macros.

In order to prevent the above problems, most experienced C programmers include parentheses around each variable in a macro and additional parentheses around the entire expression. This will allow any macro to work correctly, and it is the reason the macro named CUBE is still in error. It needs parentheses around the entire expression.

The remainder of the program is simple and will be left to your inspection and understanding.

CONDITIONAL COMPILATION - PART 1

Example program ------> **IFDEF.C**

The example program named IFDEF.C is our first illustration of a conditional compilation. **OPTION_1** is defined in line 4, and is considered defined for the entire program. Therefore when the preprocessor gets to line 6, it keeps the text between lines 6 and 8 in the program and passes it to the compiler. If **OPTION_1** was not defined when we reach line 6, the preprocessor would throw away line 7 and the compiler would never see it. Likewise line 18 is conditionally compiled based on whether **OPTION_1** is defined or not. This is a very useful construct, but not the way we are using it here. Generally it is used to include a feature if we are using a certain processor, a certain operating system, or even a special piece of hardware.

You should compile and execute the program as is, then comment out line 4 so that **OPTION_1** will not be defined, and recompile and execute the program. You will see that the extra line will not be printed because it will be thrown away by the preprocessor. Keep in mind that the preprocessor does only textual substitution or text removal and you will be able to use it effectively.

Line 26 illustrates an undefine command to the preprocessor. This removes the fact that **OPTION_1** was defined and from this point on, the program acts as though it were never defined. Of course, it does no good here since the program is completed and there are no executable statements following the undefine, but it does illustrate the undefine statement.

You should move the undefine to line 5, recompile and execute the program, and you will see that it acts as though **OPTION_1** was never defined.

CONDITIONAL COMPILATION - PART 2

Example program ------> **IFNDEF.C**

The next example program illustrates the preprocessor directive which includes code if a symbol in not defined. The ifndef directive reads literally, "if not defined", and with that much definition, its operation should be intuitive. This program will be a real exercise in logic for the diligent student, but should be understandable with a little effort. The symbol **OPTION_1** is reversed from the last program and the symbol **PRINT_DATA** is used to enable printing if it is not defined. If it is not defined, there will be some printout. This example program, much like the last one, is rather silly but illustrates the use of preprocessor directives. The next program is a little more practical.

CONDITIONAL COMPILATION - PART 3

Example program ------> **DEBUGEX.C**

The program named DEBUGEX.C is a good illustration of a very practical use of the preprocessor. In this program we define a symbol named **MY_DEBUG** at the beginning of the program. When we reach the code in the **main()** function we see why it is defined. Apparently we do not have enough information to complete this code, so we sort of slopped it in until we have a chance to talk to Bill and Linda about how to do these calculations. In the meantime, we wish to continue work on other parts of the program, so we use the preprocessor to temporarily throw away this uncompilable code for us. Because of the obnoxious message we put into line 15, it will be impossible for us to forget about the bad state of affairs

we left the code in, so we are forced to come back later and clean it up.

In this case, we are only concerned with a few lines of code, but it could be a large block of code we are working with. We could also be using this technique to handle several large blocks of code, some of which are in other modules, until Bill returns to explain the analysis and we can complete the undefined blocks.

MULTIPLE FILE PROGRAMS

For very small programs, it is expedient to include all of the code in a single file, and compile that one file for the final resulting code. It is not generally acceptable to do this because all but the most trivial programs are too big to place in a single file because the file gets to be very cumbersome to work with. It is not at all unusual for a C program to be made up of over a thousand source files. It is, of course, necessary for these files to communicate and work together as one large program.

Even though it is best not to use global variables, a variable that is defined outside of any function, it is sometimes expedient to use a few. Sometimes these variables need to be referenced by two or more different files, and C provides a way to do this. Consider the following three file portions.

```
  FILE1.C                FILE2.C                FILE3.C
int index;             extern int index;      extern int index;
extern int count;      int count;
                       static int value;      int value;
                       int main();
static void one();     void two();            void three();
```

The variable named **index** defined in FILE1.C is available to any other file for use because it is defined globally. The other two files make use of the same variable by declaring it as an **extern** variable. In essence, they are telling the compiler, "I wish to use the variable named **index** which is defined somewhere else". Anytime **index** is referred to in either of the other two files, the variable of that name is used from FILE1.C, and it can be read, or modified by any of the three files. This provides an easy way to pass data from any file to any other file, but it could lead to problems. It would be very easy for any of these files to modify **index** in some way not meant to and corrupt the data. It could be very difficult to determine which file corrupted the value of **index**.

The variable named **count** is defined in FILE2.C and referred to in the same manner defined above within FILE1.C, but is not available for use in FILE3.C because it is not declared in it. A static variable, such as **value** in FILE2.C cannot be referenced in any other file but is hidden in the declaring file by definition. A completely separate variable named **value** is defined in FILE3.C that has nothing to do with the variable of the same name in FILE2.C. In this case, FILE1.C could declare **value** as an external variable and refer to that variable in FILE3.C if desired.

The **main()** entry point can only be called by the operating system to get the program started, but the functions **two()** and **three()** can be called from anywhere within the three files because they are global functions. The function **one()** however, because it is declared static, can only be called from within the file in which it is declared. It cannot be called from within FILE2.C or FILE3.C. It is sometimes expedient to "hide" a function within a file, and it is often referred to as a local function as opposed to being a global function.

Note that some systems use only 6 characters of an external variable or function name as significant, and some are case sensitive. Check the documentation for your compiler to see if you are restrained by these limitations.

WHAT IS AN ENUMERATION VARIABLE?

Example program ------> **ENUM.C**

Load and display the program named ENUM.C for an example of how to use the **enum** type variable. Line 6 defines the first **enum** type variable named **result** which is a variable that can take on any of the values contained within the braces. Actually the variable **result** is an **int** type variable and can be assigned any of the values defined for an **int** type variable. The names within the parentheses are **int** type constants and can be used anywhere it is legal to use an **int** type constant. The constant **WIN** is assigned the value of 0, **TIE** the value 1, **BYE** the value 2, etc.

In use, the variable named **result** is used just like any **int** variable would be used as can be seen by its use in the program. The **enum** type of variable is intended to be used by you, the programmer, as a coding aid since you can use a constant named **MON** for control structures rather than the meaningless (at least to you) value of 1. Notice that **days** is assigned the values of days of the week in the remainder of the program. If you were to use a **switch** statement, it would be much more meaningful to use the labels **SUN**, **MON**, etc, rather than the more awkward 0, 1, 2, etc.

All caps are used for the enumeration values in this program as a matter of personal taste because they are all constants. There is no universal standard on this matter and each programmer is free to do as he wishes. All caps for these values tends to be standard practice however.

WHAT IS A PRAGMA?

A pragma is an instruction to your compiler to perform some particular action at compile time. Although pragmas vary from compiler to compiler and are not standardized, they perform some useful functions. Your compiler probably supports some way for you to select the optimization method by inserting a pragma into the source code. If your compiler provides a source listing file, you probably have pragmas to format the output listing to your personal preference. Check your documentation for the pragmas that are provided by your compiler.

PROGRAMMING EXERCISE

1. Write a program to count from 7 to -5 by counting down. Use **#define** statements to define the limits. (Hint, you will need to use a decrementing variable in the third part of the **for** loop control.

Return to Table of Contents

Advance to Chapter 7

Gordon Dodrill - dodrill@swcp.com - Please email any comments or suggestions.

C Tutorial - Chapter 7

STRINGS AND ARRAYS

WHAT IS A STRING?

A string is a group of characters, usually letters of the alphabet. In order to format your printout in such a way that it looks nice, has meaningful names and titles, and is aesthetically pleasing to you and the people using the output of your program, you need the ability to output text data. Actually you have already been using strings, because the second program in this tutorial, way back in Chapter 2, output a message that was handled internally as a string. A complete definition of a string is a series of **char** type data terminated by a null character.

When C is going to use a string of data in some way, either to compare it with another string, output it, copy it to another string, or whatever, the functions are set up to do what they are called to do until a null, which is a zero, is detected. Such a string is often called an ASCII-Z string. We will use a few ASCII-Z strings in this chapter.

WHAT IS AN ARRAY?

An array is a series of homogeneous pieces of data that are all identical in type, but the type can be quite complex as we will see when we get to the chapter of this tutorial discussing structures. A string is simply a special case of an array, a series of **char** type data.

Example program ------> **CHRSTRG.C**

The best way to see these principles is by use of an example, so load the program CHRSTRG.C and display it on your monitor. The first thing new is in line 6 which defines a **char** type of data entity. The square brackets define an array subscript in C, and the 5 in the brackets defines 5 data fields of type **char** all defined as part of the string variable. In the C language, all subscripts start at 0. We have 5 char type variables named, **name[0], name[1], name[2], name[3],** and **name[4]**. You must keep in mind that in C, the subscripts actually go from 0 to one less than the number defined in the definition statement. This is due to the original definition of C and these limits cannot be changed or redefined by the programmer.

HOW DO WE USE THE STRING?

The variable **name** is therefore a string which can hold up to 5 characters, but since we need room for the NULL terminating character which counts as one of the five characters, there are actually only four usable characters. To load something useful into the string, we have 5 assignment statements, each of which assigns one alphabetical character to one of the string characters. Finally, the last place in the string is filled with the numeral 0 as the end indicator and the string is complete. (A define would allow us to use the symbol NULL instead of a zero, and this would add greatly to the clarity of the program. It would be very obvious that this was a NULL and not simply a zero for some other purpose.) Now that we have the string, we will print it out with some other string data in the output statement, line 14.

The **%s** in the format portion of the **printf()** statement is the output definition used to output a string. The system will output characters starting with the first one in the string **name** until it comes to the null character, where it will quit. Notice that in the **printf()** statement, only the name of the variable which

happens to be **name** needs to be given, with no subscript since we are interested in starting at the beginning. (There is actually another reason that only the variable name is given without brackets. The discussion of that topic will be given in the next chapter.) It is important to realize that **name** by itself refers to the entire string, but **name[]** with some value in the square braces refers to only a single character in the string.

OUTPUTTING PART OF A STRING

The **printf()** in line 15 illustrates that we can output any single character of the string by using the **%c** and naming the particular character of the variable **name** we want by including the subscript. Notice that the term with the square brackets refers to only a single character in the string, so we output it with the **%c** notation which is used to format and output a single character. The last **printf()** illustrates how we can output part of the string by stating the starting point by using a subscript. The & specifies the address of **name[1]**. We will study this in the next chapter but I thought you would benefit from a little glimpse ahead, so don't worry about this construct yet.

This example may make you feel that strings are rather cumbersome to use since you have to set up each character one at a time. Strings would be very difficult to use if they had to be defined like we defined the string in this program, but we only did this so you could see the internal structure of the string. The next example program will illustrate that strings are very easy to use. Be sure to compile and execute this program.

SOME STRING FUNCTIONS

Example program ------> **STRINGS.C**

Load the example program STRINGS.C for an example of some ways to use strings. First we define four strings in lines 7 and 8. Next we come to a new function that you will find very useful, the **strcpy()** function, or string copy. It copies from one string to another until it comes to the null character in the source string. Remember that the null is actually a zero and is added to the character string by the system. It is easy to remember which one gets copied to which if you think of them like an assignment statement. Thus if you were to say, for example, `x = 23;`, the data is copied from the right entity to the left one. In the **strcpy()** function, the data is also copied from the right entity to the left, so that after execution of the first statement, the string variable **name1** will contain the string "Rosalinda", but without the double quotes, they are the compiler's way of knowing that you are defining a string. The term "Rosalinda" is actually a string constant in exactly the same way that 23 is an integer constant as used in the expression index = 23. It should be clear that line 10 is copying a string constant into a string variable.

Likewise, the string "Zeke" is copied into **name2** in line 11, then the title string is copied into the string named **title**. The title and both names are then printed out. Note that it is not necessary for the destination string to be exactly the same size as the string it will be called upon to store, only that it is at least as long as the source string plus one more character for the terminating null.

ALPHABETICAL SORTING OF STRINGS

The next function we will look at is the **strcmp()** or the string compare function illustrated in line 18. It will return a 1 if the first string is larger than the second, zero if they are the same length and have the same characters, and -1 if the first string is smaller than the second. One of the strings, depending on the result of the compare is copied into the string variable **mixed** using line 19 or 21, and the largest name

alphabetically is printed out in line 23. It should come as no surprise to you that Zeke wins because it is alphabetically larger. Length doesn't matter, only relative position in the alphabet. It might be wise to mention that the result would also depend on whether the letters were upper or lower case. There are also functions available with your C compiler to change the case of a string to all upper or all lower case if you desire. These will be used in an example program later in this tutorial.

COMBINING STRINGS

Lines 25 through 28 illustrate another new feature, the **strcat(),** or string concatenation function. This function simply adds the characters from one string onto the end of another string taking care to adjust the null so everything is still all right. In this case, **name1** is copied into **mixed**, then two blanks are concatenated to **mixed**, and finally **name2** is concatenated to the combination. The result is printed out with both names in the one string variable **mixed**.

Strings are not difficult to use and are extremely useful, but they do require some care in their use. It is an error to copy a string into a string that has been defined as shorter than the source, but the compiler will perform the copy and will overwrite a portion of your program or some other data. There is no way for the compiler to warn you of this, so you must be careful when using strings.

A quick check of the documentation for one compiler revealed about 18 string functions available for use. Some are used for copying strings with upper limits on how many characters can be copied. There are string functions to search for certain characters in a string, and others for adding characters to the front, middle, or end of a string. And of course you can remove characters from anywhere also. It would pay you to read your compiler documentation to see just what string functions are available for your use. It could greatly simplify something you will be doing in the near future if you know what is available. You should spend some time getting familiar with strings before proceeding on to the next topic. We included the file named **string.h** in line 3 because it contains prototypes for all of the string functions. A little time spent examining this file would be time well spent.

Compile and run this program and observe the results for compliance with this definition.

AN ARRAY OF INTEGERS

Example program ------> **INTARRAY.C**

Load the file INTARRAY.C and display it on your monitor for an example of an array of integers. Notice that the array is defined in much the same way we defined an array of **char** in order to do the string manipulations in the last example program. We have 12 integer variables to work with, plus one more named **index**. The names of the variables are **values[0], values[1],** ... , and **values[11]**. In lines 9 and 10 we have a loop to assign nonsense, but well defined, data to each of the 12 variables, then print all 12 out in lines 12 and 13. Note carefully that each element of the array is simply an **int** type variable capable of storing an integer value. The only difference between the variables **index** and **values[2],** for example, is in the way you address them. You should have no trouble following this program, but be sure you understand it. Compile and execute it to see if it does what you expect it to do.

AN ARRAY OF FLOATING POINT DATA

Example program ------> **BIGARRAY.C**

Load and display the program named BIGARRAY.C for an example of a program with an array of **float**

type data. This program has an extra feature to illustrate how strings can be initialized. Line 4 of the program illustrates how to initialize a string of characters. Notice that the square brackets are empty leaving it up to the compiler to count the characters and allocate enough space for our string plus the terminating null. Another string is initialized in line 11 of the body of the program but it must be declared **static** here. This prevents it from being allocated as an automatic variable and allows it to retain the string once the program is started. There is nothing else new here, the variables are assigned nonsense data and the results of all the nonsense are printed out along with a header. This program should also be easy for you to follow, so study it until you are sure of what it is doing before going on to the next topic. Once again, the **float** array can corrupt a program if it is used to write past the end of the array.

When you compile this program, you may get a few warnings of type conversions. You should have enough knowledge of C at this point to eliminate the warnings by including a few casts at the right places. If you desire, you can safely ignore the warnings.

GETTING DATA BACK FROM A FUNCTION

Example program ------> **PASSBACK.C**

Back in chapter 5 when we studied functions, I hinted to you that there was a way to get data back from a function by using an array, and that is true. Examine the program PASSBACK.C for an example of doing that. In this program, we define an array of 20 variables named **matrix** in line 8, then assign some nonsense data to the variables, and print out the first five. In line 16 we call the function **dosome()** taking along the entire array by putting the name of the array in the parentheses as the actual parameter.

The function **dosome()** beginning in line 25 has a name in its parentheses also but it prefers to call the array **list** internally. The function needs to be told that it is really getting an array passed to it and that the array is of type **int**. Line 25 does that by defining **list** as an integer type variable and including the square brackets to indicate an array. It is not necessary to tell the function how many elements are in the array. Generally a function works with an array until some end-of-data marker is found, such as a null for a string, or some other previously defined data or pattern. Many times, another piece of data is passed to the function with a count of how many elements to work with. In our present illustration, we will use a fixed number of elements to keep it simple.

So far nothing is different from the previous functions we have called except that we have passed more data points to the function this time than we ever have before, having passed 20 integer values, the entire array. We print out the first 5 again in lines 29 and 30 to see if they did indeed get passed here. In lines 32 and 33 we add ten to each of the elements and print out the new values. Finally we return to the main program and print out the same 5 data points. We find that we have modified the data stored in the calling program from within the function, and when we returned to the main program, we brought the changes back. Compile and run this program to verify this conclusion.

ARRAYS PASS DATA BOTH WAYS

We stated during our study of functions that when we passed data to a function, the system made a copy to use in the function which was thrown away when we returned. This is not the case with arrays. The actual input array is made available to the function and the function can modify it any way it wishes to. The result of the modifications will be available back in the calling program. This may seem strange to you that arrays are handled differently from single point data, but they are. It really does make sense, but you will have to wait until we get to pointers to understand it.

A HINT AT A FUTURE LESSON

Another way of getting data back from a function to the calling program is by using a pointer which we will discuss in the next chapter. When we get there we will find that the name of an array is in reality a pointer to a list of values. Don't let that worry you now, it will make sense when we get there. In the meantime, concentrate on arrays and understand the basics of them because when we get to the study of structures we will be able to define some pretty elaborate arrays.

MULTI-DIMENSIONAL ARRAYS

Example program ------> **MULTIARY.C**

Load and display the file named MULTIARY.C for an example of a program with doubly dimensioned arrays. The variable **big** is an 8 by 8 array that contains 8 times 8 or 64 elements total. The first element is **big[0][0]**, and the last is **big[7][7]**. Another array named **large** is also defined which is not square to illustrate that the array need not be square. Both are filled with data, one representing a multiplication table, and the other being formed into an addition table.

To illustrate that individual elements can be modified at will, one of the elements of **big** is assigned the value from one of the elements of **large** after being multiplied by 22 in line 17. Next **big[2][2]** is assigned the arbitrary value of 5, and this value is used for the subscripts of the assignment statement in line 19. The assignment statement in line 19 is in reality **big[5][5]** = 177; because each of the subscripts contain the value 5. This is only done to illustrate that any valid expression can be used for a subscript. It must only meet two conditions, it must be an integer (although a **char** will work just as well), and it's value must be within the range of the subscript it is being used for.

The entire matrix variable **big** is printed out in a square form in lines 21 through 26 so you can check the values to see if they did get set the way you expected them to.

PROGRAMMING EXERCISES

1. Write a program with three short strings, about 6 characters each, and use **strcpy()** to copy the string literals "one", "two", and "three" into them. Concatenate the three strings into one larger string defined with 30 characters and print the result out 10 times.
2. Define two integer arrays, each 10 elements long, called **array1** and **array2**. Using a loop, put some kind of nonsense data in each and add them term for term into another 10 element array named **arrays**. Finally, print all results in a table with an index number.
3. Define a string of some selected length, assign a word or phrase to it, and print it out as a string, then print it out as individual characters. Finally, print it out backwards by using a **for** loop with a decrementing third term. A useful function for the first term of the **for** loop is **strlen()** which returns the length of the string by counting characters up to, but not including, the terminating null.

Return to the Table of Contents

Advance to Chapter 8

Gordon Dodrill - dodrill@swcp.com - Please email any comments or suggestions.

C Tutorial - Chapter 8

POINTERS

WHAT IS A POINTER?

Example program ------> **POINTER.C**

Simply stated, a pointer is an address. Instead of being a variable, it is a pointer to a variable stored somewhere in the address space of the program. It is always best to use an example so load the file named POINTER.C and display it on your monitor for an example of a program with some pointers in it.

For the moment, ignore the data definition statement where we define **index** and two other fields beginning with a star. It is properly called an asterisk, but for reasons we will see later, let's agree to call it a star. If you observe the statement in line 8, it should be clear that we assign the value of 39 to the variable named **index**. This is no surprise, we have been doing it for several programs now. The statement in line 9 however, says to assign to **pt1** a strange looking value, namely the variable **index** with an ampersand in front of it. In this example, **pt1** and **pt2** are pointers, and the variable named **index** is a simple variable. Now we have a problem similar to the old chicken and egg problem. We need to learn how to use pointers in a program, but to do so requires that first we define the means of using the pointers in the program.

The following two rules will be somewhat confusing to you at first, but we need to state the definitions before we can use them. Take your time, and the whole thing will clear up very quickly.

TWO VERY IMPORTANT RULES

The following two rules are very important when using pointers and must be thoroughly understood.

1. A variable name with an ampersand in front of it defines the address of the variable and therefore points to the variable. You can therefore read line nine as "**pt1** is assigned the value of the address of **index**".
2. A pointer with a star in front of it refers to the value of the variable pointed to by the pointer. Line twelve of the program can be read as "The stored (starred) value to which the pointer **pt1** points is assigned the value 13". This is commonly referred to as dereferencing the pointer. Now you can see why it is convenient to think of the asterisk as a star, it sort of sounds like the word store.

MEMORY AIDS

1. Think of & as an address.
2. Think of * as a star referring to stored.

Assume for the moment that **pt1** and **pt2** are pointers (we will see how to define pointers shortly). As pointers, they do not contain a variable value but an address of a variable and can be used to point to a variable. Figure 8-1 is a graphical representation of the data space as it is configured just prior to executing line 8. A box represents a variable, and a box with a dot in it represents a pointer. At this time the pointers are not pointing at anything, so they have no arrows emanating from the boxes. Executing line 8 stores the value 39 in index.

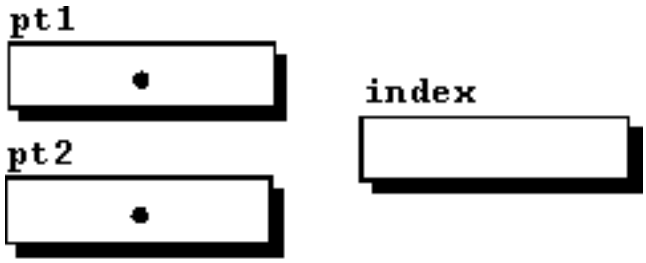

Figure 8-1

Continuing execution of the program, we come to line 9 which assigns the address of the variable **index** to the pointer **pt1** which causes **pt1** to point to **index**. Since we have a pointer to **index**, we can manipulate the value of **index** by using either the variable name itself, or the pointer. Figure 8-2 depicts the condition of the data space after executing line 9 of the program.

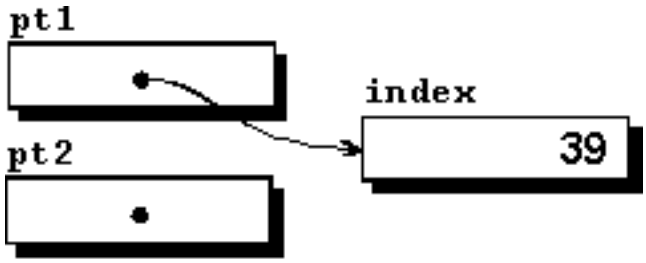

Figure 8-2

Jumping ahead a little in the program, line 12 modifies the value of **index** by using the pointer. Since the pointer **pt1** points to the variable named **index**, putting a star in front of the pointer name refers to the memory location to which it is pointing. Line 12 therefore assigns the value of 13 to **index**. Anyplace in the program where it is permissible to use the variable name **index**, it is also permissible to use the name ***pt1** since they are identical in meaning until the pointer is reassigned to some other variable.

ANOTHER POINTER

Just to add a little intrigue to the system, we have another pointer defined in this program, **pt2**. Since **pt2** has not been assigned a value prior to statement 10, it doesn't point to anything, it contains garbage. Of course, that is also true of any local variable until a value is assigned to it. The statement in line 10 assigns **pt2** the same address as **pt1**, so that now **pt2** also points to the variable named **index**. We have copied the address from one pointer to another pointer. To continue the definition from the last paragraph, anyplace in the program where it is permissible to use the variable **index**, it is also permissible to use the name ***pt2** because they are now identical in meaning. This fact is illustrated in the **printf()** statement in line 11 since this statement uses the three means of identifying the same variable to print out the same variable three times. Refer to figure 8-3 for the representation of the data space at this time.

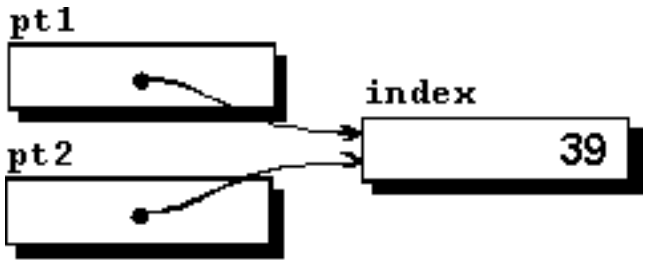

Figure 8-3

THERE IS ONLY ONE VARIABLE

Note carefully that, even though it appears that there are three variables, there is really only one variable. The two pointers each point to the single variable. This is illustrated in the statement in line 12 which assigns the value of 13 to the variable **index**, because that is where the pointer **pt1** is pointing. The **printf()** statement in line 13 causes the new value of 13 to be printed out three times. Keep in mind that there is really only one variable to be changed or printed, not three. We do have three aliases for one variable, **index**, ***pt1**, and ***pt2.** Figure 8-4 is the graphical representation of the data space at this time.

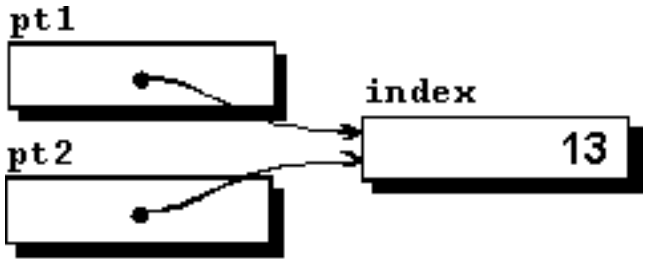

Figure 8-4

This is admittedly a very difficult concept, but since it is used extensively in all but the most trivial C programs, it is well worth your time to stay with this material until you understand it thoroughly.

HOW DO YOU DEFINE A POINTER?

Now to keep a promise and tell you how to define a pointer. Refer to line 6 of the program and you will see our old familiar way of defining the variable **index**, followed by two more definitions. The second definition can be read as "the storage location to which **pt1** points will be an **int** type variable". Therefore, **pt1** is a pointer to an **int** type variable. Likewise, **pt2** is another pointer to an **int** type variable, because it has a star (asterisk) in front of it. These two pointers can point to the same **int** variable or to two different **int** variables.

A pointer must be defined to point to a specific type of variable. Following a proper definition, it cannot be used to point to any other type of variable or it will result in a type incompatibility error.

Compile and run this program and observe that there is only one variable and the single statement in line 12 changes the one variable which is displayed three times. This material is so important that you should review it carefully if you do not fully understand it at this time. It would be a good exercise for you to draw the graphics yourself as you review the code for this program.

THE SECOND PROGRAM WITH POINTERS

Example program ------> **POINTER2.C**

In these few pages so far on pointers, we have covered a lot of territory, but it is important territory. We still have a lot of material to cover so stay in tune as we continue this important aspect of C. Load the next file named POINTER2.C and display it on your monitor so we can continue our study.

In this program we have defined several variables and two pointers. The first pointer named **there** is a pointer to a **char** type variable and the second named **pt** points to an **int** type variable. Notice also that we have defined two array variables named **strg** and **list**. We will use them to show the correspondence between pointers and array names.

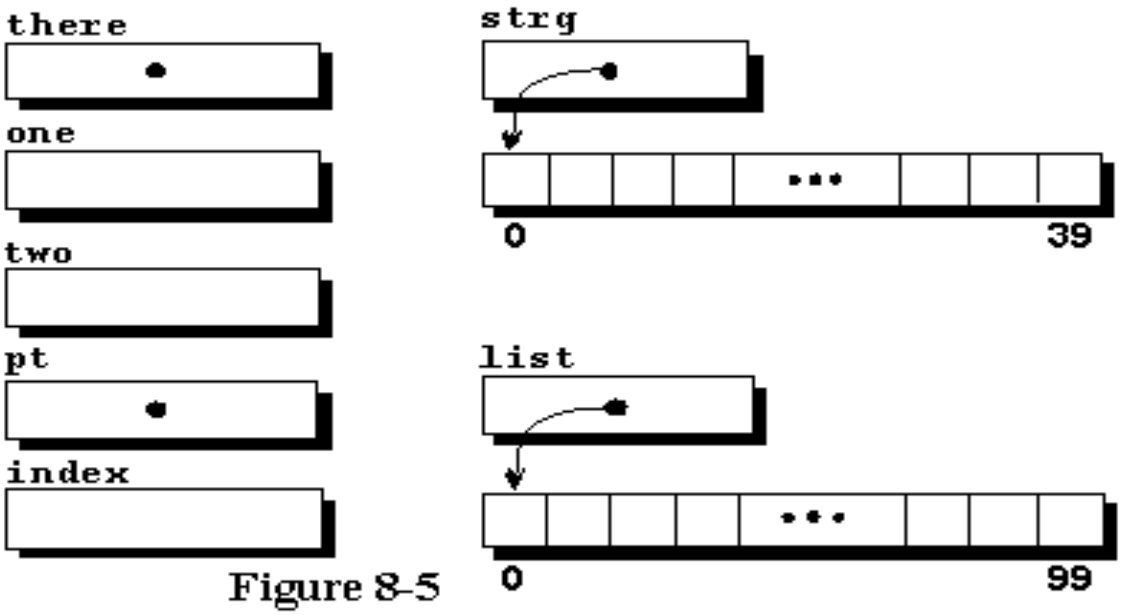

Figure 8-5

Figure 8-5 depicts the data just prior to executing line 10. There are three variables, two pointers, a string, and an array of ints, or we could say there are three variables, two pointers, and two arrays. Each array is composed of the array itself and a pointer which points to the beginning of the array according to the definition of an array in C. This will be completely defined in the next paragraph. Each array is composed of a number of identical elements of which only a few at the beginning and a few at the end are depicted graphically for convenience.

AN ARRAY NAME IS ACTUALLY A POINTER

In the C programming language, an array name is defined to be a constant pointer to the beginning of the array. This will take some explaining. Refer to the example program on your monitor. You will notice that in line 10 we assign a string constant to the string variable named **strg** so we will have some data to work with. Next, we assign the value of the first element to the variable **one**, a simple **char** variable. Next, since the string name is a constant pointer to the first element of the string, by definition of the C language, we can assign the same value to **two** by using the star and the string name (***strg**). Observe that the box with a dot pointing to a variable can be used to access the variable just like in the last program. The result of the two assignments are such that **one** now has the same value as **two**, and both contain the character T, the first character in the string. Note that it would be incorrect to write line 10 as `two = *strg[0];` because the star takes the place of the square brackets, or does the same job.

For all practical purposes, **strg** is a pointer to a **char** type variable. It does, however, have one restriction that a true pointer does not have. It cannot be changed like a variable, but must always contain the address of the first element of the string and therefore always points to the beginning of its string. It is a pointer constant. Even though it cannot be changed, it can be used to refer to other values than the one it is defined to point to, as we will see in the next section of the program.

Moving ahead to line 16, the variable **one** is assigned the value of the ninth character in the string (since the indexing starts at zero) and **two** is assigned the same value because we are allowed to index a pointer to get to values farther ahead in the string. Both variables now contain the character 'a'. Line 17 says to

add 8 to the value of the pointer **strg**, then get the value stored at that location and store it in the variable **two**.

POINTER INDEXING

The C programming language takes care of indexing for us automatically by adjusting the indexing for the type of variable the pointer is pointing to. In this case, the index of 8 is simply added to the pointer value before looking up the desired result because a **char** type variable is one byte long. If we were using a pointer to an **int** type variable, the index would be doubled and added to the pointer before looking up the value because an **int** type variable uses two bytes per value on most 16 bit microcomputers. It would multiply the index value by 4 before adding it to the pointer if an **int** used four bytes, as it does on most 32 bit systems. When we get to the chapter on structures, we will see that a variable can have many, even into the hundreds or thousands, of bytes per variable, but the indexing will be handled automatically for us by the system.

The data space is now in the state defined graphically in figure 8-6. The string named **strg** has been filled and the two variables named **one** and **two** have the letter "a" stored in them. Since the pointer variable **there** is already a pointer, it can be assigned the address of the 11th element of **strg** by the statement in line 20 of this program. Remember that since **there** is a pointer to type **char**, it can be assigned any value as long as that value represents a **char** type of address. It should be clear that the pointers must be typed in order to allow the pointer arithmetic described in the last paragraph to be done properly. The third and fourth outputs will be the same, namely the letter c.

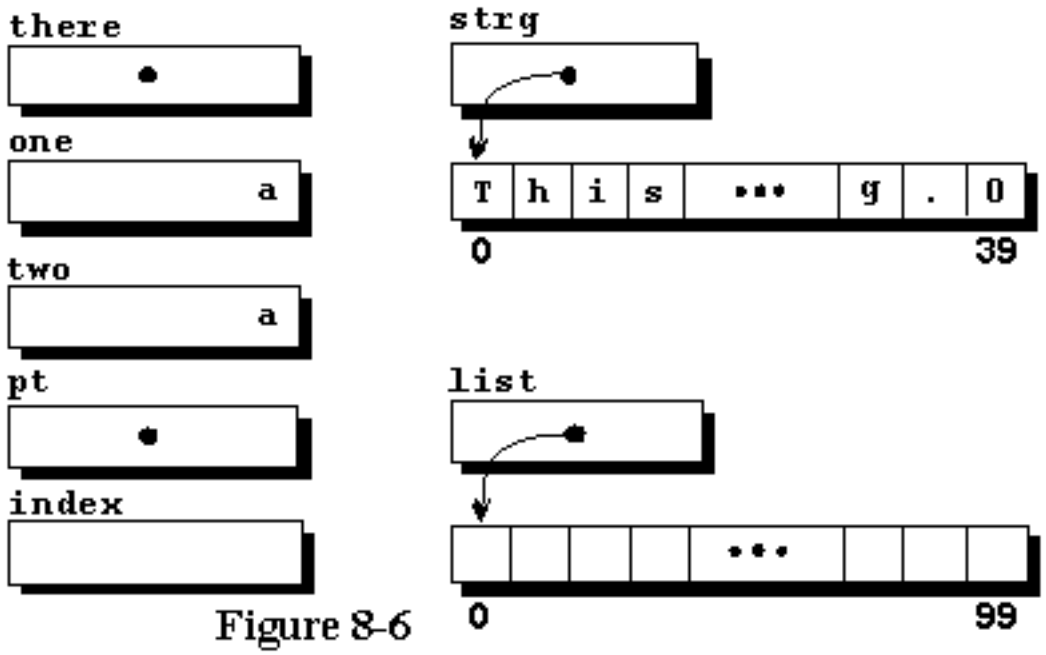

Figure 8-6

POINTER ARITHMETIC

Not all forms of arithmetic are permissible on a pointer. Only those things that make sense, considering that a pointer is an address somewhere in the computer. It would make sense to add a constant to an address, thereby moving it ahead in memory that number of places. Likewise, subtraction is permissible, moving it back some number of locations. Adding two pointers together would not make sense because absolute memory addresses are not additive. Pointer multiplication is also not allowed, as that would be a funny number. If you think about what you are actually doing, it will make sense to you what is allowed, and what is not.

NOW FOR AN INTEGER POINTER

The array named **list** is assigned a series of values from 100 to 199 in order to have some data to work with in lines 24 and 25. Next, we assign the pointer **pt** the address of the 28th element of the list and print out the same value both ways to illustrate that the system truly will adjust the index for the **int** type variable. You should spend some time in this program until you feel you fairly well understand these lessons on pointers.

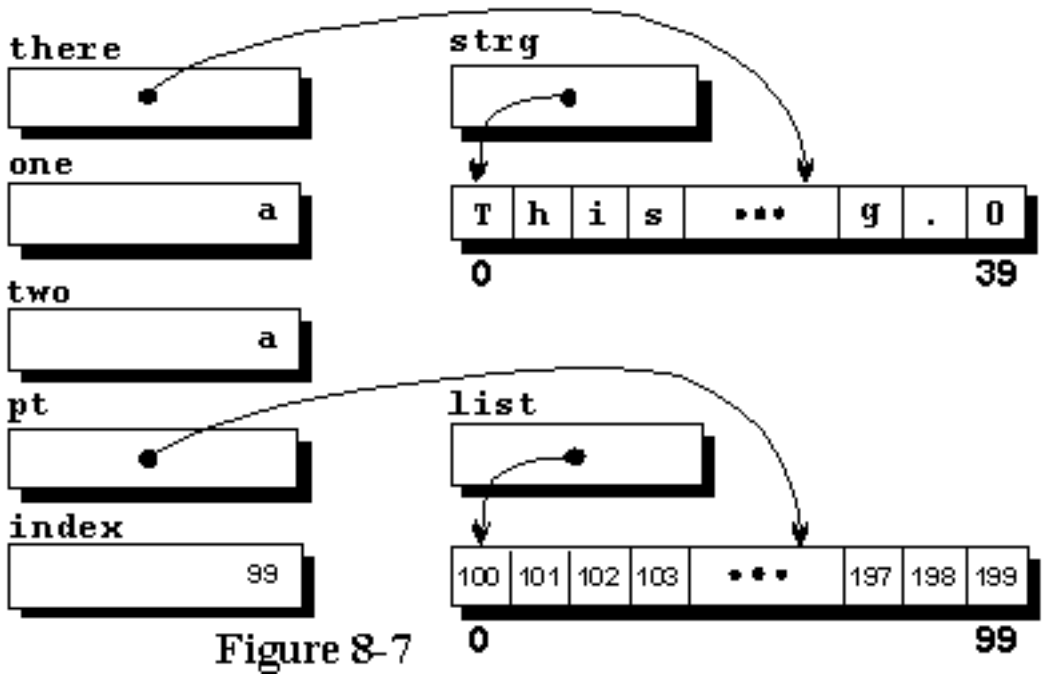

Figure 8-7

Compile and execute POINTER2.C and study the output. At the termination of execution, the data space will be as depicted in figure 8-7. Once again, it would be a good exercise for you to attempt to draw the graphic for this program as you review the code.

FUNCTION DATA RETURN WITH A POINTER

Example program ------> **TWOWAY.C**

You may recall that back in the lesson on functions we mentioned that there were two ways to get variable data back from a function. One way is through use of the array, and you should be right on the verge of guessing the other way. If your guess is through use of a pointer, you are correct. Load and display the example program named TWOWAY.C for an example of this.

In TWOWAY.C, there are two variables defined in the main program, **pecans** and **apples**. Notice that neither of these is defined as a pointer. We assign values to both of these and print them out, then call the function named **fixup()** taking both of these values along with us. The variable **pecans** is simply sent to the function, but the address of the variable **apples** is sent to the function. Now we have a problem. The two arguments are not the same, the second is a pointer to a variable. We must somehow alert the function to the fact that it is supposed to receive an integer variable and a pointer to an integer variable. This turns out to be very simple. Notice that the parameter definitions in line 23 defines **nuts** as an integer, and **fruit** as a pointer to an integer. The call in the main program therefore is now in agreement with the function heading and the program interface will work just fine.

In the body of the function, we print the two values sent to the function, then modify them and print the new values out. This should be perfectly clear to you by now. The surprise occurs when we return to the **main()** function and print out the two values again. We will find that the value of **pecans** will be restored to the value it had prior to the function call because the C language makes a copy of the item in question and takes the copy to the called function, leaving the original intact as we explained earlier. In the case of the variable **apples**, we made a copy of a pointer to the variable and took the copy of the pointer to the function. Since we had a pointer to the original variable, even though the pointer was a local copy, it pointed to the original variable and we could change the value of **apples** from within the function. When we returned to the main program, we found a changed value in **apples** when we printed it out.

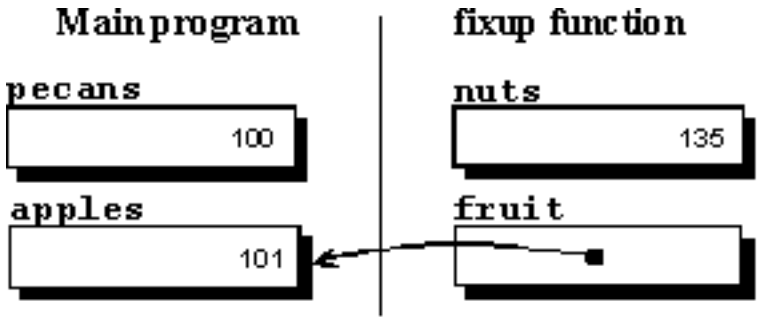

Figure 8-8

This is illustrated graphically in figure 8-8. The state of the system is illustrated following execution of line 27 of the program. The observant student will notice the prototype in line 3. This allows the compiler to check the type of both parameters when it gets to line 14 where the function is called.

By using a pointer in a function call, we can have access to the original data while excuting code within the function and change it in such a way that when we return to the calling program, we have a changed value of the original variable. In this example, there was no pointer in the main program because we simply sent the address to the function, but in many programs you will use pointers in function calls. One of the places you will find need for pointers in function calls will be when you request data input using standard input/output routines. These will be covered in the next two chapters. Compile and run TWOWAY.C and observe the output.

POINTERS ARE VALUABLE

Even though you are probably somewhat intimidated by this time about the proper use of pointers, you will find that after you gain experience, you will use them profusely in many ways. You will also use pointers in every program you write other than the most trivial because they are so useful. You should probably go over this material carefully several times until you feel comfortable with it because it is very important in the area of input/output which is next on the agenda.

A POINTER TO A FUNCTION

Example program ------> **FUNCPNT.C**

Examine the example program named FUNCPNT.C for the most unusual pointer yet. This program contains a pointer to a function, and illustrates how to use it.

Line 8 of this program defines **function_pointer** as a pointer to a function and not to just any function, it points to a function with a single formal parameter of type **float**. The function must also return nothing because of the **void** before the pointer definition. The parentheses are required around the pointer name as illustrated or the system will think it is a prototype definition for a function that returns a pointer to **void**.

You will note the prototypes given in lines 4 through 6 that declare three functions that use the same parameter and return type as the pointer. Since they are the same as the pointer, the pointer can be used to refer to them as is illustrated in the executable part of the program. Line 15 contains a call to the **print_stuff()** function, and line 16 assigns the value of **print_stuff** to **function_pointer**. Because the name of a function is defined as a pointer to that function, its name can be assigned to a function pointer variable. You will recall that the name of an array is actually a pointer constant to the first element of the array. In like manner, a function name is actually a pointer constant which is pointing to the function itself. The pointer is successively assigned the address of each of the three functions and each is called once or twice as an illustration of how a pointer to a function can be used.

A function pointer can be passed to another function as a parameter and can be used within the function to call the function which is pointed to. You are not permitted to increment or add a constant to a function pointer, it can only be assigned the value of a function with the same parameters and return with which it was initially declared. It may take you a little time to appreciate the value of this construct, but when you do understand it, you will see the flexibility built into the C programming language.

A pointer to a function is not used very often but it is a very powerful construct when needed. You should plan to do a lot of C programming before you find a need for this technique. I mention it here only to prevent you being unduly intimidated by this difficult concept. We will continue to study pointers by examining their use in additional example programs.

PROGRAMMING EXERCISES

1. Define a character array and use **strcpy()** to copy a string into it. Print the string out by using a loop with a pointer to print out one character at a time. Initialize the pointer to the first element and use the double plus sign to increment the pointer. Use a separate integer variable to count the characters to print.
2. Modify the program from programming exercise 1 to print out the string backwards by pointing to the end and using a decrementing pointer.

Return to Table of Contents

Advance to Chapter 9

Gordon Dodrill - dodrill@swcp.com - Please email any comments or suggestions.

C Tutorial - Chapter 9

STANDARD INPUT/OUTPUT

THE STDIO.H HEADER FILE

Example program ------> **SIMPLEIO.C**

Examine the file SIMPLEIO.C for our first look at a file with standard I/O. Standard I/O refers to the places where most data is either read from, the keyboard, or written to, the video monitor. Since they are used so much, they are used as the default I/O devices and do not need to be named in the Input/Output instructions. This will make more sense when we actually start to use them so let's look at the file in front of you.

The first thing you should take notice of is the second line of the example file, the line with `#include <stdio.h>`. This is very much like the **#define** we have already studied, except that instead of a simple substitution, an entire file is read in at this point. The system will find the file named stdio.h and read its entire contents in, replacing this statement. Obviously then, the file named stdio.h must contain valid C source statements that can be compiled as part of a program. You will recall that we stated earlier that the preprocessor does textual substitution. This particular file is composed of several standard #defines and prototypes to define some of the standard I/O operations. The file is called a header file and you will find several different header files on the source disks that came with your C compiler. Each of the header files has a specific purpose and any or all of them can be included in any program. Most header files contain definitions of a few types, function prototypes for the functions in its group, and some macros.

Your C compiler uses the double quote marks to indicate that the search for the include file will begin in the current directory, and if it not found there, the search will continue in the include directory as set up in the environment for your compiler. It also uses the "less than" and "greater than" signs to indicate that the file search should begin in the directory specified in the environment. Most of the programs in this tutorial use the "<" and ">" in the include statements. The next program uses the double quotes to illustrate the usage. Note that this will result is a slightly slower (but probably unnoticeable) compilation because the system will search the current directory first. If you know the include file is not in the current directory, it is best to use the "<" and ">" with the filename.

As many includes can be used as necessary, and it is perfectly all right for one header file to include one or more additional header files. It is very common to include four or five header files in a program.

It would be a profitable exercise for you to inspect the header file limits.h at this time for a complete definition of the sizes of all simple variables on your system. You should be able to understand most of this file by this point in your study of C.

INPUT/OUTPUT OPERATIONS IN C

Actually the C programming language has no input or output operations defined as part of the language, they must be user defined. Since everybody does not want to reinvent his own input and output operations, the compiler writers have done a lot of this for us and supplied us with several input functions and several output functions to aid in our program development. The functions have become a standard, and you will find the same functions available in every compiler. In fact, the industry standard of the C

language definition has become the book written by Kernigan and Ritchie, and they have included these functions in their definition.

Occasionally, when reading literature about C, you will find an author refer to K & R. This refers to the book, "The C Programming Language", written by Kernigan and Ritchie. You would be advised to purchase a copy for reference. The second edition of this book is available and is definitely the preferred edition. Note that the book by Kernigan and Ritchie does not cover the ANSI-C standard, but it is still the preferred book for a general reference to the C programming language. The major item that is not covered is the use of prototypes, but you can easily integrate your knowledge of prototypes into the concise descriptions of other C constructs given in this book. Consider it a source of information after you gain some experience with C because it is not a very good book to learn the language from.

You should print out the file named stdio.h and spend some time studying it. There will be a lot that you will not understand about it, but parts of it will look familiar. The name stdio.h is sort of cryptic for "standard input/output header", because that is exactly what it is. It defines the standard input and output functions in the form of #defines, macros, and prototypes for the functions. Don't worry too much about the details of this now. You can always return to this topic later for more study if it interests you, but you will really have no need to completely understand the STDIO.H file. You will have a tremendous need to use it however, so these comments on its use and purpose are necessary.

OTHER INCLUDE FILES

When you begin writing larger programs and splitting them up into separately compiled portions, you will have occasion to use some definitions common to each of the portions. It would be to your advantage to make a separate file containing the definitions and use the **#include** to insert it into each of the files. If you want to change any of the common statements, you will only need to change one file and you will be assured of having all of the common statements agree. This is getting a little ahead of ourselves but you now have an idea how the **#include** directive can be used with your own files.

BACK TO THE FILE NAMED SIMPLEIO.C

Let's continue our tour of the file in question. The one variable named **c** is defined and a message is printed out with the familiar **printf()** function. We then find ourselves in a continuous loop as long as the value of **c** is not equal to capital X. If there is any question in your mind about the loop control, you should review chapter 3 before continuing. The two new functions within the loop are of paramount interest in this program since they are new functions to us. These are functions to read a character from the keyboard and display a character on the monitor.

The function **getchar()** reads a single character from the standard input device, the keyboard, and assigns it to the variable named **c**. The next function **putchar()**, uses the standard output device, the video monitor, and outputs the character contained in the variable named **c**. The character is output at the current cursor location and the cursor is advanced one space for the next character. The system is therefore taking care of a lot of the overhead for us. The loop continues reading and displaying characters until we type a capital X which terminates the loop.

Compile and run this program for a few surprises. When you type on the keyboard, you will notice that what you type is displayed faithfully on the screen, and when you hit the return key, the entire line is repeated. We only told it to output each character once but it seems to be saving the characters up and redisplaying them. A short explanation is in order.

THE OPERATING SYSTEM IS HELPING US OUT

We need to understand a little bit about how the operating system works to understand what is happening here. When data is read from the keyboard, under control of the operating system, the characters are stored in a buffer until a carriage return is entered at which time the entire string of characters is given to the program. When the characters are being typed, however, the characters are displayed one at a time on the monitor. This is called echo, and happens in many of the applications you run.

With the above paragraph in mind, it should be clear that when you are typing a line of data into SIMPLEIO, the characters are being echoed by the operating system, and when you return the carriage by hitting return or enter, the characters are given to the program. As each character is given to the program, it displays it on the screen resulting in a repeat of the line typed in. To better illustrate this, type a line with a capital X somewhere in the middle of the line. You can type as many characters as you like following the X and they will all display because the characters are being read in by the operating system, echoed to the monitor, and placed in the input buffer. The operating system doesn't think there is anything special about a capital X. When the string is given to the program, however, the characters are accepted by the program one at a time and sent to the monitor one at a time, until a capital X is encountered. After the capital X is displayed, the loop is terminated, and the program is terminated. The characters on the input line following the capital X are not displayed because the capital X signalled program termination.

Compile and run SIMPLEIO.C. After running the program several times and feeling confident that you understand the above explanation, we will go on to another program.

Don't get discouraged by the above seemingly weird behavior of the I/O system. It is strange, but there are other ways to get data into the computer. You will actually find the above method useful for many applications, and you will find some of the following useful also.

ANOTHER STRANGE I/O METHOD

Example program ------> **SINGLEIO.C**

Load the file named SINGLEIO.C and display it on your monitor for another method of character I/O. Once again, we start with the standard I/O header file using the double quote method of defining it. Then we define a variable named **c**, and we print a welcoming message. Like the last program, we are in a loop that will continue to execute until we type a capital X, but the action is a little different here.

Note that **conio.h** and the **_getch()** function described below are not a part of the ANSI-C standard but are available on most C compilers written for DOS.

The function named **_getch()** is a get character function. It differs from the function named **getchar()** in that it does not get tied up in DOS. It reads the character in without echo, and puts it directly into the program where it is operated on immediately. This function therefore reads a character, immediately displays it on the screen, and continues the operation until a capital X is typed. Note that although **_getch()** is available with most popular microcomputer C compilers, it is not included in the ANSI standard and may not be available with all C compilers. It's use may therefore make a program nonportable. If your compiler does not support the **_getch()** function, you can simply ignore this example program.

When you compile and run this program, you will find that there is no repeat of the lines when you hit a carriage return, and when you hit the capital X, the program terminates immediately. No carriage return is needed to get it to accept the line with the X in it, so this program operates a little differently from the last one. However, we do have another problem here, since there is no linefeed with the carriage return.

NOW WE NEED A LINE FEED

Example program ------> **BETTERIN.C**

It is not apparent to you in most application programs but when you hit the enter key, the program supplies a linefeed to go with the carriage return. You need to return to the left side of the monitor and you also need to drop down a line. The linefeed is not automatic. We need to improve our program to do this also. If you will load and display the program named BETTERIN.C, you will find a change to incorporate this feature.

In BETTERIN.C, we have two additional statements at the beginning that will define the character codes for the linefeed (LF), and the carriage return (CR). If you look at any ASCII table you will find that the codes 10 and 13 are exactly as defined here. In the main program, after outputting the character in line 16, we compare it to CR, and if it is equal to CR, we also output a linefeed which is the LF. We could have completely omitted the two **#define** statements and used the statement `if (c == 13) putchar(10);` but it would not be very descriptive of what we are doing here. The method used in this program represents better programming practice.

You will notice that line 17 deviates from the usual style for an **if** statement, but we have a choice. We can format the code any way we desire to improve the readability. It is strictly a programmer's choice.

Compile and run BETTERIN.C to see if it does what we have said it should do. It should display exactly what you type in, including a linefeed with each carriage return, and should stop immediately when you type a capital X. If your compiler does not support **_getch()**, use the **getchar()** function.

WHICH METHOD IS BEST?

We have examined two methods of reading characters into a C program, and are faced with a choice of which one we should use. It really depends on the application because each method has advantages and disadvantages.

When using the first method, the operating system is actually doing all of the work for us by storing the characters in an input buffer and signaling us when a full line has been entered. We could write a program that, for example, did a lot of calculations, then went to get some input. While we were doing the calculations, the operating system would be accumulating a line of characters for us, and they would be there when we were ready for them. However, we could not read in single keystrokes because the operating system would not report a buffer of characters to us until it recognized a carriage return.

The second method, used in BETTERIN.C, allows us to get a single character, and act on it immediately. We do not have to wait until the operating system decides we can have a line of characters. We cannot do anything else while we are waiting for a character because we are waiting for the input keystroke and tying up the entire machine. This method is useful for highly interactive types of program interfaces. It is up to you as the programmer to decide which is best for your needs.

I should mention at this point that there is also an **_ungetch()** function that works with the **_getch()**

function and is also not a part of the ANSI-C standard library, but is available with most DOS compilers. If you **_getch()** a character and find that you have gone one too far, you can **_ungetch()** it back to the input device. This simplifies some programs because you don't know that you don't want the character until you get it. You can only **_ungetch()** one character back to the input device, but that is sufficient to accomplish the task this function was designed for. It is difficult to demonstrate this function in a simple program so its use will be up to you to study when you need it. Another function that may be available with your compiler, but is not part of the ANSI standard, is the **_getche()** function which is identical to the **_getch()** function except that it echoes the character to the monitor for you.

The discussion so far in this chapter should be a good indication that, while the C programming language is very flexible, it does put a lot of responsibility on you as the programmer to keep many details in mind.

NOW TO READ IN SOME INTEGERS

Example programs ------> **INTIN.C**

Load and display the file named INTIN.C for an example of reading some formatted data from the keyboard. The structure of this program is very similar to the last three except that we define an **int** type variable and loop until the variable somehow acquires the value of 100.

Instead of reading in a character at a time, as we have in the last three example programs, we read in an entire integer value with one call using the function named **scanf()**. This function is very similar to the **printf()** that you have been using for quite some time by now except that it is used for input instead of output. Examine the line with the **scanf()** and you will notice that it does not ask for the variable **valin** directly, but gives the address of the variable since it expects to have a value returned from the function. Recall that a function must have the address of a variable in order to return a value to that variable in the calling program. Failing to supply a pointer to the parameter in the **scanf()** function is the most common problem encountered in using this function.

The function **scanf()** scans the input line until it finds the first data field. It ignores leading blanks and in this case, it reads integer characters until it finds a blank or an invalid decimal character, at which time it stops reading and returns the value.

Remembering our discussion above about the way the input buffer works, it should be clear that nothing is actually acted on until a complete line is entered and it is terminated by a carriage return. At this time, the buffer is input, and our program will search across the line reading all integer values it can find until the line is completely scanned. This is because we are in a loop and we tell it to find a value, print it, find another, print it, etc. If you enter several values on one line, it will read each one in succession and display the values. Entering the value of 100 will cause the program to terminate, and entering the value 100 with other values following, will cause termination before the following values are considered.

IT MAKES WRONG ANSWERS SOMETIMES

If your system uses a 2 byte integer and you enter a number up to and including 32767, it will display correctly, but if you enter a larger number, it will appear to make an error. For example, if you enter the value 32768, it will display the value of -32768, entering the value 65536 will display as a zero. These are not errors but are caused by the way an **int** variable is defined. The most significant bit of the 16 bit pattern available for the integer variable is the sign bit, so there are only 15 bits left for the value. The variable can therefore only have the values from -32768 to 32767, any other values are outside the range

of integer variables. This is up to you to take care of in your programs. It is another example of the increased responsibility you must assume using C rather than another high level language such as Pascal, Modula-2, etc.

The above paragraph is true for 16 bit C compilers. There is an ever increasing possibility that your compiler uses an integer value stored in a field size larger than 16 bits. If that is the case, the same principles will be true but with different limits than those given above.

Compile and run this program, entering several numbers on a line to see the results, and with varying numbers of blanks between the numbers. Try entering numbers that are too big to see what happens, and finally enter some invalid characters to see what the system does with nondecimal characters.

CHARACTER STRING INPUT

Example program ------> **STRINGIN.C**

Load and display the file named STRINGIN.C for an example of reading a string variable from the keyboard. This program is identical to the last one except that instead of an integer variable, we have defined a string variable with an upper limit of 24 characters (remember that a string variable must have a null character at the end). The variable in the **scanf()** does not need an & because **big** is an array variable and by definition it is already a pointer. This program should require no additional explanation. Compile and run it to see if it works the way you expect.

You probably got a surprise when you ran it because it separated your sentence into separate words. When used in the string mode of input, **scanf()** reads characters into the string until it comes to either the end of a line or a blank character. Therefore, it reads a word, finds the blank following it, and displays the result. Since we are in a loop, this program continues to read words until it exhausts the input buffer. We have written this program to stop whenever it finds a capital X in column 1, but since the sentence is split up into individual words, it will stop anytime a word begins with capital X. Try entering a 5 word sentence with a capital X as the first character in the third word. You should get the first three words displayed, and the last two simply ignored when the program stops.

Try entering more than 24 characters to see what the program does. In an actual program, it is your responsibility to count characters and stop when the input buffer is full. You may be getting the feeling that a lot of responsibility is placed on you when writing in C. Along with this responsibility you get a lot of flexibility in the bargain also. Because **scanf()** has no way to stop when the input array is full, it should not be used for string input in a quality program. It was used here only as an illustration of input programming.

INPUT/OUTPUT PROGRAMMING IN C

C was not designed to be used as a language for lots of input and output, but as a systems language where a lot of internal operations are required. You would do well to use another language for I/O intensive programming, but C could be used if you desire. The keyboard input is very flexible, allowing you to get at the data in a very low level way, but very little help is given you. It is therefore up to you to take care of all of the bookkeeping chores associated with your required I/O operations. This may seem like a real pain in the neck, but in any given program, you only need to define your input routines once and then use them as needed. Don't let this worry you. As you gain experience with C, you will easily handle your I/O requirements.

One final point must be made about these I/O functions. It is perfectly permissible to intermix **scanf()** and **getchar()** functions during read operations. In the same manner, it is also fine to intermix the output functions, **printf()** and **putchar()** in any way you desire.

IN MEMORY I/O

Example program ------> **INMEM.C**

The next operation may seem a little strange at first, but you will probably see lots of uses for it as you gain experience. Load the file named INMEM.C and display it for another type of I/O, one that never accesses the outside world, but stays in the computer.

In INMEM.C, we define a few variables, then assign some values to the ones named **numbers** for illustrative purposes and then use an **sprintf()** function. The function acts just like a normal **printf()** function except that instead of printing the line of output to a device, it prints the line of formatted output to a character string in memory. In this case the string goes to the string variable named **line**, because that is the string name we inserted as the first argument in the **sprintf()** function. The spaces after the 2nd **%d** were put there to illustrate that the next function will search properly across the line. We print the resulting string and find that the output is identical to what it would have been by using a **printf()** instead of the **sprintf()** in the first place. You will see that when you compile and run the program shortly.

Since the generated string is still in memory, we can now read it with the function **sscanf()**. We tell the function in its first argument that **line** is the string to use for its input, and the remaining parts of the line are exactly what we would use if we were going to use the **scanf()** function and read data from some input device. Note that it is essential that we use pointers to the data because we want to return data from a function. Just to illustrate that there are different ways to declare a pointer, two methods are used, but all are ultimately pointers. The first two simply declare the address of the elements of the array, while the last three use the fact that **result**, without the accompanying subscript, is a pointer. Just to keep it interesting, the values are read back in reverse order. Finally, the values are displayed on the monitor.

IS THAT REALLY USEFUL?

It seems sort of silly to read input data from within the computer but it does have a real purpose. It is possible to read data from an input device using any of the standard functions and then do a format conversion in memory. You could read in a line of data, look at a few significant characters, then use these formatted input routines to reduce the line of data to internal representation. That would sure beat writing your own data formatting routines.

STANDARD ERROR OUTPUT

Example program ------> **SPECIAL.C**

Sometimes it is desirable to redirect the output from the standard output device to a file. However, you may still want the error messages to go to the standard output device, in our case the monitor. This next function allows you to do that. Load and display SPECIAL.C for an example of this new function.

The program consists of a loop with two messages output, one to the standard output device and the other to the standard error device. The message to the standard error device is output with the function **fprintf()** and includes the device name **stderr** as the first argument. Other than those two small changes,

it is the same as our standard **printf()** function. (You will see more of the **fprintf()** function in the next chapter, but its operation fit in better as a part of this chapter.) Ignore the line with the **exit** for the moment, we will return to it.

Compile and run this program, and you will find 12 lines of output on the monitor. To see the difference, run the program again with redirected output to a file named STUFF by entering the following line at the operating system prompt;

```
C:> special >stuff
```

This time you will only get the 6 lines output to the standard error device, and if you look in your directory, you will find that the file named STUFF contains the other 6 lines, those to the standard output device. You can use I/O redirection with any of the programs we have run so far, and as you may guess, you can also read from a file using I/O redirection but we will study a better way to read from a file in the next chapter. More information about I/O redirection can be found in your operating system manual.

WHAT ABOUT THE exit(4) STATEMENT?

Now to keep our promise about the **exit(4)** statement. Redisplay the file named SPECIAL.C on your monitor. The last statement exits the program and returns the value of 4 to the operating system. Any number within a predefined range can be used within the parentheses for communication with your operating system. If you are operating in a DOS environment and executing code in a BATCH file, this number can be tested with the ERRORLEVEL command.

Most compilers that operate in several passes return a 1 with this mechanism to indicate that a fatal error has been detected and it would be a waste of time to go on to another compilation pass resulting in even more errors. A return value of 0 would indicate that no error was detected.

PROGRAMMING EXERCISES

1. Write a program to read in a character using a loop, and display the character in its normal **char** form. Also display it as a decimal number. Check for a dollar sign to use as the stop character. Use the **_getch()** form of input so it will print immediately. Hit some of the special keys, such as function keys, when you run the program for some surprises. You will get two inputs from the special keys, the first being a zero which is the indication to the system that a special key was hit.
2. Add a character string to SINGLEIO.C and store the input characters in the string. When the X is detected, add a terminating null to the string and print out the string with a **printf()** function call.

Return to Table of Contents

Advance to Chapter 10

Gordon Dodrill - dodrill@swcp.com - Please email any comments or suggestions.

C Tutorial - Chapter 10

FILE INPUT/OUTPUT

OUTPUT TO A FILE

Example program ------> **FORMOUT.C**

Load and display the file named FORMOUT.C for your first example of writing data to a file. We begin as before with the include statement for stdio.h, and include the header for the string functions. Then we define some variables for use in the example including a rather strange looking new type.

The type **FILE** is a structure (we will study structures in the next chapter) and is defined in the stdio.h file. It is used to define a file pointer for use in file operations. The definition of C requires a pointer to a **FILE** type to access a file, and as usual, the name can be any valid variable name. Many writers use **fp** for the name of this first example file pointer so I suppose we should start with it too.

OPENING A FILE

Before we can write to a file, we must open it. What this really means is that we must tell the system that we want to write to a file and what the filename is. We do this with the **fopen()** function illustrated in line 11 of the program. The file pointer, **fp** in our case, will point to the structure for the file and two arguments are required for this function, the filename first, followed by the file attribute. The filename is any valid filename for your operating system, and can be expressed in upper or lower case letters, or even mixed if you so desire. It is enclosed in double quotes. For this example we have chosen the name TENLINES.TXT. This file should not exist on your disk at this time. If you have a file with this name, you should change its name or move it because when we execute this program, its contents will be overwritten. If you don't have a file by this name, this program will create one and write some data into it.

Note that we are not forced to use a string constant for the file name as we have done here. This is only done here for convenience. We can use a string variable which contains the filename then use any method we wish to fill in the name of the file to open. This will be illustrated later in this chapter.

READING ("r")

The second parameter is the file attribute and can be any of three letters, "r", "w", or "a", and must be lower case. There are actually additional attributes available in C to allow more flexible I/O, and after you complete your study of this chapter, you should check the documentation for your compiler to study the additional file opening attributes. When an "r" is used, the file is opened for reading, a "w" is used to indicate a file to be used for writing, and an "a" indicates that you desire to append additional data to the data already in an existing file. Opening a file for reading requires that the file already exist. If it does not exist, the file pointer will be set to NULL and can be checked by the program. It is not checked in this program, but could be easily checked as follows.

```
if (fp == NULL) {
   printf("File failed to open\n");
   exit (1);
}
```

Good programming practice would dictate that all file pointers be checked to assure proper file opening in a manner similar to the above code. The value of 1 used as the parameter of **exit()** will be explained shortly.

WRITING ("w")

When a file is opened for writing, it will be created if it does not already exist and it will be reset if it does, resulting in deletion of any data already there. If the file fails to open for any reason, a NULL will be returned so the pointer should be tested as above.

APPENDING ("a")

When a file is opened for appending, it will be created if it does not already exist and it will be initially empty. If it does exist, the data input point will be set to the end of the data already contained in the file so that new data will be added to any data that already exists in the file. Once again, the return value can and should be checked for proper opening.

OUTPUTTING TO THE FILE

The job of actually outputting to the file is nearly identical to the outputting we have already done to the standard output device. The only real differences are the new function names and the addition of the file pointer as one of the function arguments. In the example program, **fprintf()** replaces our familiar **printf()** function name, and the file pointer defined earlier is the first argument within the parentheses. The remainder of the statement looks like, and in fact is identical to, the **printf()** statement.

CLOSING A FILE

To close a file, use the function **fclose()** with the file pointer in the parentheses. Actually, in this simple program, it is not necessary to close the file because the system will close all open files before returning to the operating system. It would be good programming practice for you to get in the habit of closing all files in spite of the fact that they will be closed automatically, because that would act as a reminder to you of what files are open at the end of each program.

You can open a file for writing, close it, and reopen it for reading, then close it, and open it again for appending, etc. Each time you open it, you could use the same file pointer, or you could use a different one. The file pointer is simply a tool that you use to point to a file and you decide what file it will point to.

Compile and run this program. When you run it, you will not get any output to the monitor because it doesn't generate any. After running it, look in your current directory for a file named TENLINES.TXT and examine it's contents. That is where your output will be. Compare the output with that specified in the program. It should agree. If you add the pointer test code described above, and if the file couldn't be opened for any reason, there will be one line of text on the monitor and the file will be empty.

Do not erase the file named TENLINES.TXT yet. We will use it in some of the other examples in this chapter.

OUTPUTTING A SINGLE CHARACTER AT A TIME

Example program ------> **CHAROUT.C**

Load the next example file, CHAROUT.C, and display it on your monitor. This program will illustrate how to output a single character at a time.

The program begins with the include statements, then defines some variables including a file pointer. The file pointer is named **point** this time, but we could have used any other valid variable name. We then define a string of characters to use in the output function using a **strcpy()** function. We are ready to open the file for appending and we do so with the **fopen()** function, except this time we use the lower cases for the filename. This is done simply to illustrate that some operating systems don't care about the case of the filename. Some operating systems, including UNIX, are case sensitive for filenames, so you will need to fix the case before compiling and executing this program. Notice that the file will be opened for appending so we will add to the lines inserted during the last program. If the file could not be opened properly, a NULL value is returned by the **fopen()** function.

Lines 14 through 18 check to see if the file opened properly and returns an error indication to the operating system if it did not. The constant named EXIT_FAILURE is defined in the stdlib.h file and is usually defined to have the value of 1. The constant named EXIT_SUCCESS is also defined in the stdlib.h file and is usually defined to have the value of 0. The operating system can use the returned value to determine if the program operated normally and can take appropriate action if neccessary. For example, if a two part program is to be executed and the first part returns an error indication, there is no need to execute the second part of the program. Your compiler probably executes in several passes with each successive pass depending on successful completion of the previous pass.

The program is actually two nested **for** loops. The outer loop is simply a count to ten so that we will go through the inner loop ten times. The inner loop calls the function **putc()** repeatedly until a character in the string named **others** is detected to be a zero. This is the terminating null for the string.

THE putc() FUNCTION

The part of the program we are interested in is the **putc()** function in line 23. It outputs one character at a time, the character being the first argument in the parentheses and the file pointer being the second and last argument. Why the designer of C made the pointer first in the **fprintf()** function, and last in the **putc()** function is a good question for which there may be no answer. It seems like this would have been a good place to have used some consistency.

When the textline **others** is exhausted, a newline is needed because a newline was not included in the definition above. A single **putc()** is then executed which outputs the **\n** character to return the carriage and do a linefeed.

When the outer loop has been executed ten times, the program closes the file and terminates. Compile and run this program but once again there will be no output to the monitor. You need to assure that TENLINES.TXT is in the current directory prior to execution.

Following execution of the program, examine the contents of the file named TENLINES.TXT and you will see that the 10 new lines were added to the end of the 10 that already existed. If you run it again, yet another 10 lines will be added. Once again, do not erase this file because we are still not finished with it.

READING A FILE

Example program ------> **READCHAR.C**

Load the file named READCHAR.C and display it on your monitor. This is our first program which can read from a file. This program begins with the familiar include statements, some data definitions, and the file opening statement which should require no explanation except for the fact that an "r" is used here because we want to read from this file. In this program, we check to see that the file exists, and if it does, we execute the main body of the program. If it doesn't exist, we print a message and quit. If the file does not exist, the system will set the pointer equal to NULL which we test in line 12. If the pointer is NULL we display a message and terminate the program.

The main body of the program is one **do while** loop in which a single character is read from the file and output to the monitor until an EOF (end of file) is detected from the input file. The file is then closed and the program is terminated.

CAUTION CAUTION CAUTION

At this point, we have the potential for one of the most common and most perplexing problems of programming in C. The variable returned from the **getc()** function is a character, so we can use a **char** variable for this purpose. There is a problem that could develop here if we happened to use an **unsigned char** however, because C returns a minus one for an EOF. An **unsigned char** type variable is not capable of containing a negative value. An **unsigned char** type variable can only have the values of zero to 255, so it will return a 255 for a minus one which can never compare to the EOF. This is a very frustrating problem to try to find. The program can never find the EOF and will therefore never terminate the loop. This is easy to prevent. Always use a **int** type variable when the return can be an EOF, because an **int** is always signed. According to the ANSI-C standard, a **char** can be implemented as either a signed or an unsigned type by any particular compiler.

Some compilers use a char type that is not 8 bits long. If your compiler uses other than 8 bits for a char type variable, the same arguments apply. Do not use an unsigned type if you need to check for an EOF returned by the function, because an EOF is usually defined as -1 which cannot be returned in an unsigned type variable.

There is yet another problem with this program but we will worry about it when we get to the next program and solve it with the one following that.

After you compile and run this program and are satisfied with the results, it would be a good exercise to change the name of TENLINES.TXT and run the program again to see that the NULL test actually works as stated. Be sure to change the name back because we are still not finished with TENLINES.TXT. In a real production program, you would not actually terminate the program. You would give the user the opportunity to enter another filename for input. We are interested in illustrating the basic file handling techniques here, so we are using a very simple error handling method.

READING A WORD AT A TIME

Example program ------> **READTEXT.C**

Load and display the file named READTEXT.C for an example of how to read a word at a time. This program is nearly identical to the last except that this program uses the **fscanf()** function to read in a string at a time. Because the **fscanf()** function stops reading when it finds a space or a newline character, it will read a word at a time, and display the results one word to a line. You will see this when you compile and run it, but first we must examine a programming problem.

It is left as an exercise for the student to include a check for proper file opening and performing a meaningful response if it does not open. A meaningful response is to simply output an error message and exit to the operating system.

THIS IS A PROBLEM

Inspection of the program will reveal that when we read data in and detect the EOF, we print out something before we check for the EOF resulting in an extra line of printout. What we usually print out is the same thing printed on the prior pass through the loop because it is still in the buffer named **oneword**. We therefore must check for EOF before we execute the **printf()** function. This has been done in READGOOD.C, which you will shortly examine, compile, and execute.

Compile and execute the program we have been studying, READTEXT.C and observe the output. If you haven't changed TENLINES.TXT you will end up with "Additional" and "lines." on two separate lines with an extra "lines." displayed at the end of the output because of the **printf()** before checking for EOF. Note that some compilers apparently clear the buffer after printing so you may get an extra blank line instead of two lines with "lines." on them.

Notice that we failed to check that the file opened properly. This is very poor practice, and it will be left as an exercise for you to add the required code to do so in a fashion similar to that used in the READCHAR.C example program.

NOW LET'S FIX THE PROBLEM

Example program ------> **READGOOD.C**

Compile and execute READGOOD.C and observe that the extra "lines." does not get displayed because of the extra check for the EOF in the middle of the loop. This was also the problem referred to when we looked at READCHAR.C, but I chose not to expound on it there because the error in the output was not so obvious.

Once again there is no check for the file opening properly, but you know how to fix it by now and you should do so as an exercise.

We should point out that an experienced C programmer would not write the code as given in this example because it compares **c** to EOF twice during each pass through the loop and this is inefficient. We have been using code that works and is very easy to understand, but as you gain experience with C, you will begin to use more efficient coding methods, even if they tend to become harder to read and understand. An experienced C programmer would code lines 12 through 17 of READGOOD.C in the following manner;

```
while((c = fscanf(fp1, "%s", oneword) != EOF)
{
   printf("%s\n", oneword);
}
```

There is no question that this code is more difficult to read, but if you spend some time studying it, you will find that it is identical to the code in the example program. Even though is is more efficient, it is not clear whether the slight gain in efficiency is worth the reduced readability. If the program saves ten milliseconds when reading in a file once a day, and it takes a programmer an hour longer to make a

modification to the code a year after the program is released, there is not much savings in using such code. Even though the time assumptions are all judgement calls in the above text, it is plain to see that there are often tradeoffs when writing a program. You will make many decisions concerning execution efficiency and readability when you are writing non-trivial programs.

This is a rather contrived example, because most experienced C programmers would not think this code is at all cryptic, but is written in a standard C notation. As you gain experience, you will come to accept this as clearly written C code. The philosophical argument about code complexity and readability has been made however, and should be considered for all software development.

FINALLY, WE READ A FULL LINE

Example program ------> **READLINE.C**

Load and display the file READLINE.C for an example of reading a complete line. This program is very similar to those we have been studying except that we read a complete line in this example program.

We are using **fgets()** which reads an entire line, including the newline character, into a buffer. The buffer to be read into is the first argument in the function call, and the maximum number of characters to read is the second argument, followed by the file pointer. This function will read characters into the input buffer until it either finds a newline character, or it reads the maximum number of characters allowed minus one. It leaves one character for the end of string null character. In addition, if it finds an EOF, it will return a value of NULL. In our example, when the EOF is found, the pointer named **c** will be assigned the value of NULL. NULL is defined as zero in your stdio.h file.

When we find that the pointer named **c** has been assigned the value of NULL, we can stop processing data, but we must check before we print just like in the last program. Last of course, we close the file.

HOW TO USE A VARIABLE FILENAME

Example propgram ------> **ANYFILE.C**

Load and display the program ANYFILE.C for an example of reading from any file. This program asks the user for the filename desired, and reads in the filename, storing it in a string. Then it opens that file for reading. The entire file is then read and displayed on the monitor. It should pose no problems to your understanding so no additional comments will be made.

Compile and run this program. When it requests a filename, enter the name and extension of any text file available, even one of the example C programs.

Enter an invalid file name to see what the system does when it cannot open the file. If you were a user of this program, and possibly not very computer literate, would you prefer that the program gave you a cryptic message of some sort, or would you prefer that the program displayed a neat message such as "The file you have asked for is not available. Would you like to enter another filename?". Of course this is the message that you can emit when you find that the file did not open properly. Your users will appreciate the effort you put into error handling for their program.

HOW DO WE PRINT?

Example program ------> **PRINTDAT.C**

Load the last example program in this chapter, the one named PRINTDAT.C for an example of how to print. This program should not present any surprises to you, so we will move very quickly through it.

Once again, we open TENLINES.TXT for reading and we open PRN for writing. Printing is identical to writing data to a disk file except that we use a standard name for the filename. Many C compilers use the reserved filename of PRN that instructs the compiler to send the output to the printer. There are other names that are used occasionally such as LPT, LPT1, or LPT2. Check the documentation for your particular compiler. Some of the newest compilers use a predefined file pointer such as **stdprn** for the print file. Once again, check your documentation.

The program is simply a loop in which a character is read, and if it is not the EOF, it is displayed and printed. When the EOF is found, the input file and the printer output files are both closed. Note that good programming practice includes checking both file pointers to assure that the files opened properly. You can now erase TENLINES.TXT from your disk. We will not be using it in any of the later chapters.

A READING ASSIGNMENT

Spend some time studying the documentation for your compiler and reading about the following functions. You will not understand everything about them but you will get a good idea of how the library functions are documented.

fopen(), fclose(), putc(), putchar(), printf(), fprintf(), scanf(), fgets()

Also spend some time studying stdio.h, looking for prototypes for the above functions and for the declaration of FILE.

PROGRAMMING EXERCISES

1. Write a program that will prompt for a filename for an input file, prompt for a filename for a write file, and open both plus a file to the printer. Enter a loop that will read a character, and output it to the file, the printer, and the monitor. Stop at EOF.
2. Prompt for a filename to read. Read the file a line at a time and display it on the monitor with line numbers.
3. Modify ANYFILE.C to test if the file exists and print a message if it doesn't. Use a method similar to that used in READCHAR.C.

Return to Table of Contents

Advance to Chapter 11

Gordon Dodrill - dodrill@swcp.com - Please email any comments or suggestions.

C Tutorial - Chapter 11

STRUCTURES AND UNIONS

WHAT IS A STRUCTURE?

Example program ------> **STRUCT1.C**

A structure is a user defined data type. Using a structure you have the ability to define a new type of data considerably more complex than the types we have been using. A structure is a combination of several different previously defined data types, including other structures we have defined. A simple definition is, "a structure is a grouping of related data in a way convenient to the programmer or user of the program." The best way to understand a structure is to look at an example, so if you will load and display STRUCT1.C, we will do just that.

The program begins with a structure definition. The keyword **struct** is followed by three simple variables between the braces, which are the components of the structure. After the closing brace, you will find two variable names listed, **boy**, and **girl**. According to the definition of a structure, **boy** is now a variable composed of three elements, **initial**, **age**, and **grade**. Each of the three fields are associated with **boy**, and each can store a variable of its respective type. The variable named **girl** is also a variable containing three fields with the same names as those of **boy** but are actually different variables. We have therefore defined 6 simple variables, but they are grouped into 2 variables of a structure type.

A SINGLE COMPOUND VARIABLE

Let's examine the variable named **boy** more closely. As stated above, each of the three elements of **boy** are simple variables and can be used anywhere in a C program where a variable of their type can be used. For example, the **age** element is an **int** variable and can therefore be used anywhere in a C program where it is legal to use an **int** type variable, in calculations, as a counter, in I/O operations, etc. We now have the problem of defining how to use the simple variable named **age** which is a part of the compound variable named **boy**. To do so we use both names with a decimal point between them with the major name first. Thus **boy.age** is the complete variable name for the **age** field of **boy**. This construct can be used anywhere in a C program that it is desired to refer to this field. In fact, it is illegal to use the name **boy** or **age** alone because they are only partial definitions of the complete field. Alone, the names refer to nothing. (Actually the name **boy** alone does have meaning when used with a modern C compiler. We will discuss this later.)

ASSIGNING VALUES TO THE VARIABLES

Using the above definition, we can assign a value to each of the three fields of **boy** and each of the three fields of **girl**. Note carefully that **boy.initial** is actually a **char** type variable, because it was defined as one in the structure, so it must be assigned a character of data. In line 12, **boy.initial** is assigned the character R in agreement with the above rules. The remaining two fields of **boy** are assigned values in accordance with their respective types. Finally the three fields of **girl** are

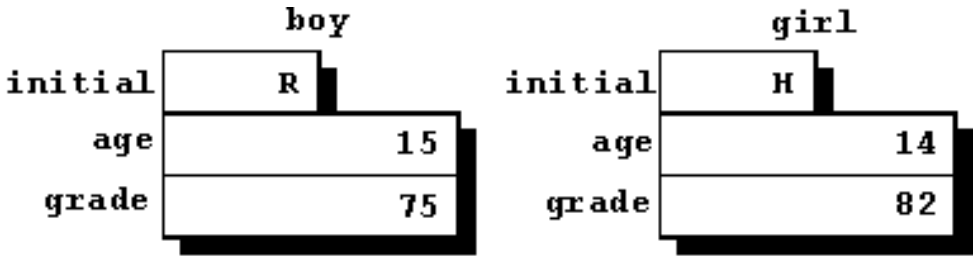

Figure 11-1

assigned values but in a different order to illustrate that the order of assignment is not critical. You will notice that we used the value of the boy's age when we defined the girl's age. This illustrates the use of one member of the structure. Figure 11-1 is a graphical representation of the data following execution of line 18.

HOW DO WE USE THE RESULTING DATA?

Now that we have assigned values to the six simple variables, we can do anything we desire with them. In order to keep this first example simple, we will simply print out the values to see if they really do exist as assigned. If you carefully inspect the **printf()** statements, you will see that there is nothing special about them. The compound name of each variable is specified because that is the only valid name by which we can refer to these variables.

Structures are a very useful method of grouping data together in order to make a program easier to write and understand. This first example is too simple to give you even a hint of the value of using structures, but continue on through these lessons and eventually you will see the value of using structures. Compile and run STRUCT1.C and observe the output.

AN ARRAY OF STRUCTURES

Example program ------> **STRUCT2.C**

Load and display the next program named STRUCT2.C. This program contains the same structure definition as before but this time we define an array of 12 variables named **kids**. It should be clear that this program contains 12 times 3 = 36 simple variables, each of which can store one item of data provided that it is of the correct type. We also define a simple variable named **index** for use in the **for** loops.

In order to assign each of the fields a value, we use a **for** loop and each pass through the loop results in assigning a value to each of the fields of one structure variable. One pass through the loop assigns all of the values for one of the kids. This would not be a very useful way to assign data in a real situation, but a loop could read the data in from a file and store it in the correct fields in a real application. You might consider this the crude beginning of a data base, which it is.

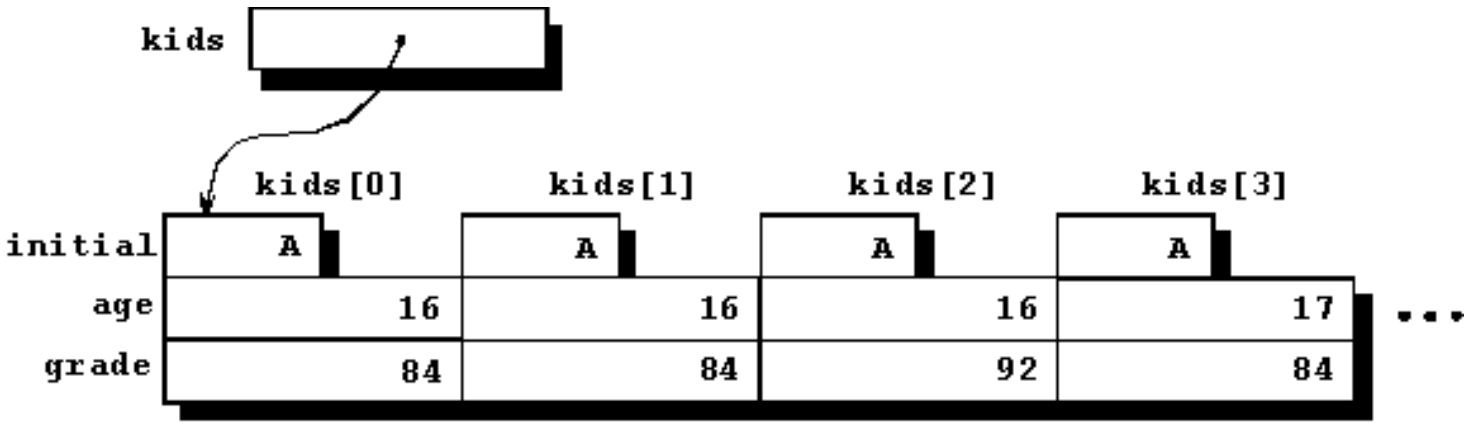

Figure 11-2

In the next few instructions of this program we assign new values to some of the fields to illustrate the method used to accomplish this. It should be self explanatory, so no additional comments will be given. Figure 11-2 is a graphical representation of the data for this program following execution of line 25.

A RECENT UPGRADE TO THE C LANGUAGE

All good C compilers will allow you to copy an entire structure with one statement. This was not always permitted in the C language but it is a part of the ANSI standard, so you should feel free to use it with your C compiler if it is available. Line 27 is an example of using a structure assignment. In this statement, all 3 fields of **kids[4]** are copied into their respective fields of **kids[10]**.

WE FINALLY DISPLAY ALL OF THE RESULTS

The last few statements contain a **for** loop in which all of the generated values are displayed in a formatted list. Compile and run the program to see if it does what you expect it to do. You will need to remove line 27 if your compiler does not support structure assignments.

USING POINTERS AND STRUCTURES TOGETHER

Example program ------> **STRUCT3.C**

Examine the file named STRUCT3.C for an example of using pointers with structures. This program is identical to the last program except that it uses pointers for some of the operations.

The first difference shows up in the definition of variables following the structure definition. In this program we define a pointer named **point** which is defined as a pointer that points to the structure. It would be illegal to try to use this pointer to point to any other variable type. There is a very definite reason for this restriction in C as we have alluded to earlier and will review in the next few paragraphs.

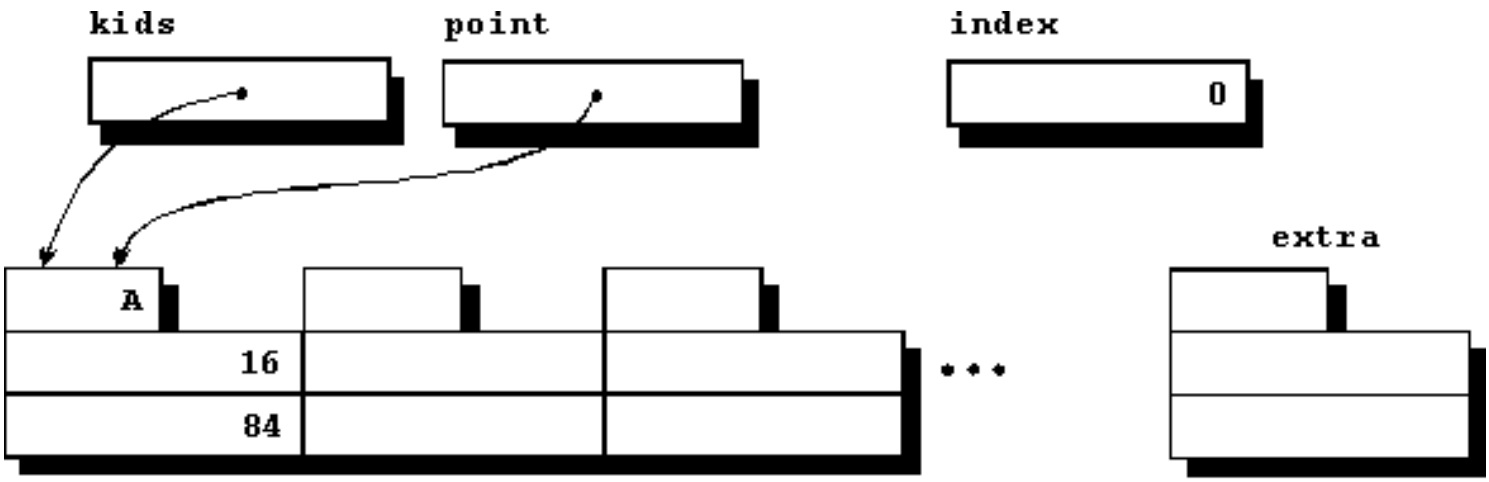

Figure 11-3

The next difference is in the **for** loop where we use the pointer for accessing the data fields. Recall from chapter 8 of this tutorial that we said that the name of an array is actually a pointer to the first element of the array. Since **kids** is a pointer constant that points to the first element of the array which is a structure, we can define **point** in terms of **kids**. The element named **kids** is a constant so it cannot be changed in value, but **point** is a pointer variable and can be assigned any value consistent with its being required to point to the structure. If we assign the value of **kids** to **point** then it should be clear that **point** will also point to the first element of the array, a structure containing three fields. Figure 11-3 is a graphical representation of the data space following the first pass through the loop starting in line 16.

POINTER ARITHMETIC

Adding 1 to **point** will now cause it to point to the second field of the array because of the way pointers are handled in C. The system knows that the structure contains three variables and it knows how many memory elements are required to store the complete structure. Therefore if we tell it to add one to the pointer, it will actually add the number of memory elements required to get to the next element of the array. If, for example, we were to add 4 to the pointer, it would advance the value of the pointer 4 times the size of the structure, resulting in it pointing 4 elements farther along the array. This is the reason a pointer cannot be used to point to any data type other than the one for which it was defined.

Now to return to the program displayed on your monitor. It should be clear from the previous discussion that as we go through the loop, the pointer will point to one of the array elements each time. We can therefore use the pointer to reference the various elements of each of the structures as we go through the loop. Referring to the elements of a structure with a pointer occurs so often in C that a special method of notation was devised. Using `point->initial` is the same as using `(*point).initial` which is really the way we did it in the last two programs. Remember that ***point** is the stored data to which the pointer points and the construct should be clear. The "->" is made up of the minus sign and the greater than sign. You will find experienced C programmers using this pointer dereference profusely when you read their code in magazines and other publications.

Since the pointer points to the structure, we must once again define which of the elements we wish to refer to each time we use one of the elements of the structure. There are, as we have seen, several different methods of referring to the members of the structure. When executing the **for** loop used for output at the end of the program, we use three different methods of referring to the structure elements. This would be considered very poor programming practice, but is done this way here to illustrate to you that they all lead to the same result. This program will probably require some study on your part to fully understand, but it will be worth your time and effort to grasp these principles.

Lines 34 and 35 are two additional examples of structure assignment which do nothing useful, but are included here for your benefit. Compile and run this program, and once again, if your compiler does not support structure assignment, you will need to remove lines 34 and 35.

NESTED AND NAMED STRUCTURES

Example program ------> **NESTED.C**

Examine the file named NESTED.C for an example of a nested structure. The structures we have seen so far have been very simple, although useful. It is possible to define structures containing dozens and even hundreds or thousands of elements but it would be to the programmers advantage not to define all of the elements at one pass but rather to use a hierarchical structure definition. This will be illustrated with the program on your monitor.

The first structure contains three elements but is followed by no variable name. We therefore have not defined any variables, only a structure, but since we have included a name at the beginning of the structure, the structure is named **person**. The name **person** can be used to refer to the structure but not to any variable of this structure type. It is therefore a new type that we have defined, and we can use the new type in the same way we use **int**, **char**, or any other types that exist in C. The only restriction is that this new name must always be associated with the keyword **struct**.

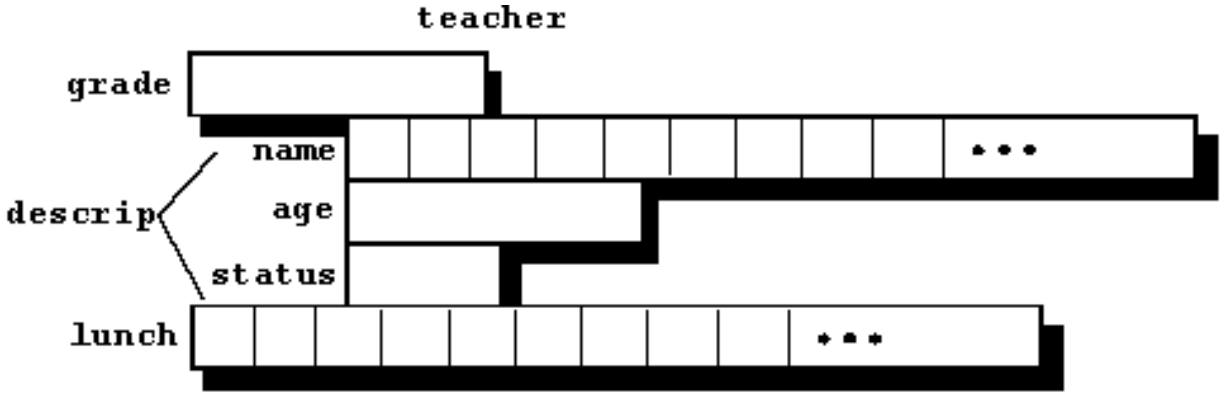

Figure 11-4

The next structure definition contains three fields with the middle field being the previously defined structure which we named **person**. The variable which has the type of **person** is named **descrip**. So the new structure contains two simple variables, **grade** and a string named **lunch**, and the structure named **descrip**. Since **descrip** contains three variables, the new structure actually contains 5 variables. This structure is also given a name **alldat**, which is another type definition. Finally, within the **main()** function, we define an array of 53 variables each with the structure defined by the type **alldat**, and each with the name **student**. If that is clear, you will see that we have defined a total of 53 times 5 variables, each of which is capable of storing a value.

Since we have a new type definition we can use it to define two more variables. The variables **teacher** and **sub** are defined in line 23 to be variables of the type **alldat**, so that each of these two variables contain 5 fields in which we can store data. Figure 11-4 is a graphical representation of the variable named **teacher** after it is defined in line 23.

NOW TO USE SOME OF THE FIELDS

In lines 25 through 29 of the program, we will assign values to each of the fields of **teacher**. The first field is the **grade** field and is handled just like the other structures we have studied because it is not part of the nested structure. Next we wish to assign a value to her **age** which is part of the nested structure. To address this field we start with the variable name **teacher** to which we append the name of the group **descrip**, and then we must define which field of the nested structure we are interested in, so we append the variable name **age**. The teachers **status** is handled in exactly the same manner as her **age**, but the last two fields are assigned strings using the string copy function **strcpy()** which must be used for string assignment. Notice that the variable names in the **strcpy()** function are still variable names even though they are made up of several parts each. We included the string.h header file in line 2 so we could call the string copy function.

The variable **sub** is assigned nonsense values in much the same way, but in a different order since they do not have to occur in any required order. Finally, a few of the student variables are assigned values for illustrative purposes and the program ends. None of the values are printed for illustration since several were printed in the last examples.

Compile and run this program, but when you run it, you may get a stack overflow error. C uses its own internal stack to store the automatic variables, but some C compilers predefine a stack as small as 2048 bytes as a default. This program requires more than that for the defined structures so it will be necessary for you to increase the stack size. Consult your compiler documentation for details concerning the method of increasing the stack size. There is no standard way to do this. There is another way around this problem, and that is to move the variable definitions outside of the main program where they will be external variables and will not be

allocated on the stack. The result is that they will not be kept on the internal stack and the stack will not overflow. It would be good experience for you to try both methods of fixing this problem.

MORE ABOUT STRUCTURES

It is possible to continue nesting structures until you get totally confused. If you define them properly, the computer will not get confused because there is no stated limit as to how many levels of nesting are allowed. There is probably a practical limit of three beyond which you may get confused, but the language has no limit. In addition to nesting, you can include as many structures as you desire in any level of structures, such as defining another structure prior to **alldat** and using it in **alldat** in addition to using **person**. The structure named **person** could be included in **alldat** two or more times if desired, as could pointers to it.

Structures can contain arrays of other structures which in turn can contain arrays of simple types or other structures. It can go on and on until you lose all reason to continue. I am only trying to illustrate to you that structures are very valuable and you will find them great aids to programming if you use them wisely. Be conservative at first, and get bolder as you gain experience. Keep in mind that a structure is designed to group related data together.

More complex structures will not be illustrated here, but you will find examples of additional structures in the example programs included in the last chapter of this tutorial. For example, see the include file named VC.H on the distribution disk.

WHAT ARE UNIONS?

Example program ------> **UNION1.C**

Examine the file named UNION1.C for an example of a union. Simply stated, a union allows you a way to look at the same data with different types, or to use the same data with different names.

In this example we have two elements to the union, the first part being the integer named **value**, which is stored as a two byte variable somewhere in the computers memory. The second element is made up of two character variables named **first** and **second**. These two variables are stored in the same storage locations that **value** is stored in, because that is what a union does. A union allows you to store different types of data in the same physical storage locations. In this case, you could put an integer number in **value**, then retrieve it in its two halves by getting each half using the two names **first** and **second**. This technique is often used to pack data bytes together when you are, for example, combining bytes to be used in the registers of the microprocessor.

Accessing the fields of the union are very similar to accessing the fields of a structure and will be left to you to determine by studying the example.

One additional note must be given here about the program. When it is run using some C compilers, the data will be displayed with leading f's due to the hexadecimal output promoting the **char** type variables to **int** and extending the sign bit to the left. Converting the **char** type data fields to **int** type fields prior to display should remove the leading f's from your display. This will involve defining two new **int** type variables and assigning the **char** type variables to them. This will be left as an exercise for you. Note that the same problem will come up in a few of the later files in this tutorial.

Compile and execute this program and observe that the data is displayed as an **int** and as two **char** variables. The **char** variables may be reversed in order because of the way an **int** variable is stored internally in your computer. If your system reverses these variables, don't worry about it. It is not a problem but it can be a very interesting area of study if you are so inclined.

ANOTHER UNION EXAMPLE

Example program ------> **UNION2.C**

Examine the file named UNION2.C for another example of a union, one which is much more common. Suppose you wished to build a large database including information on many types of vehicles. It would be silly to include the number of propellers on a car, or the number of tires on a boat. In order to keep all pertinent data, however, you would need those data points for their proper types of vehicles. In order to build an efficient data base, you would need several different types of data for each vehicle, some of which would be common, and some of which would be different. That is exactly what we are doing in the example program on your monitor.

In this program, we will define a complete structure, then decide which of the various types can go into it. We will start at the top and work our way down. First, we define a few constants with the #defines, and begin the program itself. We define a structure named **automobile** containing several fields which you should have no trouble recognizing, but we define no variables at this time.

A NEW CONCEPT, THE TYPEDEF

Next we define a new type of data with a **typedef**. This defines a complete new type that can be used in the same way that **int** or **char** can be used. Notice that the structure has no name, but at the end where there would normally be a variable name there is the name BOATDEF. We now have a new type, BOATDEF, that can be used to define a structure anyplace we would like to. Notice that this does not define any variables, only a new type. Using all caps for the name is a personal preference only and is not a C standard but is used by many experienced C programmers. It makes the **typedef** look different from a variable name.

We finally come to the big structure that defines our data using the building blocks already defined above. The structure is composed of 5 parts, two simple variables named **vehicle** and **weight**, followed by the union, and finally the last two simple variables named **value** and **owner**. Of course the union is what we need to look at carefully here, so focus on it for the moment. You will notice that it is composed of four parts, the first part being the variable **car** which is a structure of the type that we defined previously. The second part is a variable named **boat** which is a structure of the type BOATDEF previously defined. The third part of the union is the variable **airplane** which is a structure defined in place in the union. Finally we come to the last part of the union, the variable named **ship** which is another structure of the type BOATDEF.

I hope it is obvious to you that all four could have been defined in any of the three ways shown, but the three different methods were used to show you that any could be used. In practice, the clearest definition would probably have occurred by using the **typedef** for each of the parts.

WHAT DO WE HAVE NOW?

We now have a structure that can be used to store any of four different kinds of data structures. The size of every record will be the size of that record containing the largest union. In this case part 1 is the largest union because it is composed of three integers, the others being composed of an integer and a character each. The first member of this union would therefore determine the size of all structures of this type. The resulting structure can be used to store any of the four types of data, but it is up to the programmer to keep track of what is stored in each variable of this type. The variable named **vehicle** was designed into this structure to keep track of the type of vehicle stored here. The four defines at the top of the page were designed to be used as indicators stored in the variable named **vehicle**.

A few examples of how to use the resulting structure are given in the next few lines of the program. Some of the variables are defined and a few of them are printed out for illustrative purposes.

The union is not used too frequently, and almost never by beginning programmers. You will encounter it occasionally so it is worth your effort to at least know what it is. You do not need to know the details of it at this time, so don't spend too much time studying it. When you do have a need for a variant structure, a union, you can learn it at that time. For your own benefit, however, do not slight the structure. You should use the structure often.

WHAT IS A BITFIELD?

Example program ------> **BITFIELD.C**

Load and display the program named BITFIELD.C for an example of how to define and use a bitfield. In this program, we have a union made up of a single **int** type variable in line 6 and the structure defined in lines 7 through 12. The structure is composed of three bitfields named **x**, **y**, and **z**. The variable named **x** is only one bit wide, the variable **y** is two bits wide and adjacent to the variable **x**, and the variable **z** is two bits wide and adjacent to **y**. Moreover, because the union causes the bits to be stored in the same memory location as the variable **index**, the variable **x** is the least significant bit of the variable **index**, **y** is the next two bits, and **z** is stored in the next two bits of **index**.

Compile and run the program and you will see that as the variable **index** is incremented by one each time through the loop, and you will see the bitfields of the union counting due to their respective locations within the integer definition.

One thing must be pointed out, the bitfields must be defined as parts of an **unsigned int** or your compiler will issue an error message.

WHAT IS THE BITFIELD GOOD FOR?

The bitfield is very useful if you have a lot of data to separate into individual bits or groups of bits. Many systems use some sort of a packed format to get lots of data stored in a few bytes. Your imagination is your only limitation to the efficient use of this feature of C.

MORE STYLE ISSUES

Example program ------> **STYLE3.H**

Examine the file named STYLE3.H for our first example of a header file that really looks like one. You will notice several constant declarations, a few structure declarations, and some prototypes. Nothing in this file generates anything that uses memory, since there are no variables defined and no code is defined here. Each of those use some memory, but all of the constructs in this file do nothing but create declarations which are then used by other portions of the program. This header file, if it is general enough, can be used by many different implementations.

Spend a few minutes and observe the style. Take notice especially of the order of the various entities. The constants are defined first, followed by the structures since they generally use one or more of the constants. Finally, the prototypes are defined since they often make use of one or more of the structures in their parameter lists or their return values.

Example program ------> **STYLE3.C**

Examine the file named STYLE3.C which uses some of the definitions in the header STYLE3.H header file. The observant student will notice that not everything that is defined in the header file is used in this implementation file, and it really doesn't need to be. Since a header file is meant to be general purpose, all things within the file will not be used every time the header file itself is used in a program.

In this case, the structure named **alldat** is used in lines 13 and 14, after being included here in line 9. The rest of the program is written in exactly the same manner that it was written when we defined the structure locally. This program can be compiled and executed just like all of the other programs in this tutorial.

PROGRAMMING EXERCISES

1. Define a named structure containing a string for a name, an integer for feet, and another for arms. Use the new type to define an array of about 6 items. Fill the fields with data and print them out as follows.

```
A human being has 2 legs and 2 arms.
A dog has 4 legs and 0 arms.
A television set has 4 legs and 0 arms.
A chair has 4 legs and 2 arms.
```

2. Rewrite exercise 1 using a pointer to print the data out.

Return to Table of Contents

Advance to Chapter 12

Gordon Dodrill - dodrill@swcp.com - Please email any comments or suggestions.

C Tutorial - Chapter 12

DYNAMIC ALLOCATION

WHAT IS DYNAMIC ALLOCATION?

Example program ------> **DYNLIST.C**

Dynamic allocation is very intimidating to a person the first time he comes across it, but that need not be. Simply relax and read this chapter carefully and you will have a good grounding in a very valuable programming resource. All of the variables in every program up to this point have been static variables as far as we are concerned. (Actually, some of them have been automatic and were dynamically allocated for you by the system, but it was transparent to you.) In this chapter, we will study some dynamically allocated variables. They are variables that do not exist when the program is loaded, but are created dynamically as they are needed while the program is running. It is possible, using these techniques, to create as many variables as needed, use them, and deallocate their memory space for reuse by other variables. As usual, the best teacher is an example, so examine the program named DYNLIST.C.

We begin by defining a named structure, **animal**, with a few fields pertaining to dogs. We do not define any variables of this type, only three pointers. If you search through the remainder of the program, you will find no variables defined, so we have nothing to store data in. All we have to work with are three pointers, each of which are capable of pointing to variables of the defined structure named **animal**. In order to do anything, we need some variables, so we will create some dynamically.

DYNAMIC VARIABLE CREATION

The statement in line 16, which assigns something to the pointer **pet1** will create a dynamic structure containing three variables. The heart of the statement is the **malloc()** function buried in the middle of the statement. This is a memory allocate function that needs the rest of the code in this line to completely define it. The **malloc()** function, by default, will allocate a piece of memory on a heap that is "n" characters in length and will be of type character. The "n" must be specified as the only argument to the function. We will discuss "n" shortly, but first we need to define the heap.

WHAT IS A HEAP?

A heap is a predefined area of memory which can be accessed by the program to store data and variables. The data and variables are allocated on the heap by the system as calls to **malloc()** are made. The system keeps track of where the data is stored. Data and variables can be deallocated as desired, leading to holes in the heap. The system knows where the holes are and will use them for additional data storage as more **malloc()** calls are made. The structure of the heap is therefore a very dynamic entity, changing constantly.

BACK TO THE MALLOC FUNCTION

Hopefully the very brief description of the heap and the overall plan for dynamic allocation helped you to understand what we are doing with the **malloc()** function. It simply asks the system for a block of memory of the size specified, and returns a pointer which is pointing to the first element of the block. The only argument in the parentheses is the size of the block desired and in our present case, we desire a

block that will hold one of the structures we defined at the beginning of the program. The **sizeof** operator is new, new to us at least. It returns the size in bytes of the argument within its parentheses. It therefore, returns the size of the structure named **animal**, in bytes, and that number is used as a parameter of the **malloc()** call. At the completion of that call, we have a block on the heap allocated to us, with the pointer named **pet1** pointing to the block of data.

WHAT IF malloc() FAILS?

If there is not enough memory available to supply you with the block of data requested, **malloc()** does not return a valid pointer but instead returns the value of NULL. The return value should always be checked before attempting to use it, but we are ignoring it in this program for two reasons. First, and foremost, there is a huge load of information already introduced here and we wish to keep it as simple as possible and study C in byte-sized chunks. And secondly, we are asking for a tiny amount of memory in this example program, so there should be no problem with allocating it.

Keep in mind that all dynamically allocated memory must be carefully checked in any meaningful program, and to illustrate it, we will be very careful to check the return value in the next two example programs as an illustration to you of exactly how to do it.

WHAT IS A CAST?

We still have a funny looking construct at the beginning of the **malloc()** function call, which is called a cast. The **malloc()** function returns a pointer to type **void** by default. You really cannot use a pointer to **void**, so it must be changed to some other type. You can define the pointer type with the construct given on the example line. In this case we want the pointer to point to a structure of type **animal**, so we tell the compiler with this cast. Even if you omit the cast, most compilers will return a pointer correctly, give you a warning, and go on to produce a working program. It is better programming practice to provide the compiler with the cast to prevent getting a warning message.

The data space of the computer is depicted graphically by figure 12-1 following execution of line 16. The graphical notation defines the pointer as pointing to the structure. As far as the program is concerned, the pointer is actually pointing to all three members taken as a group rather than to only the first element.

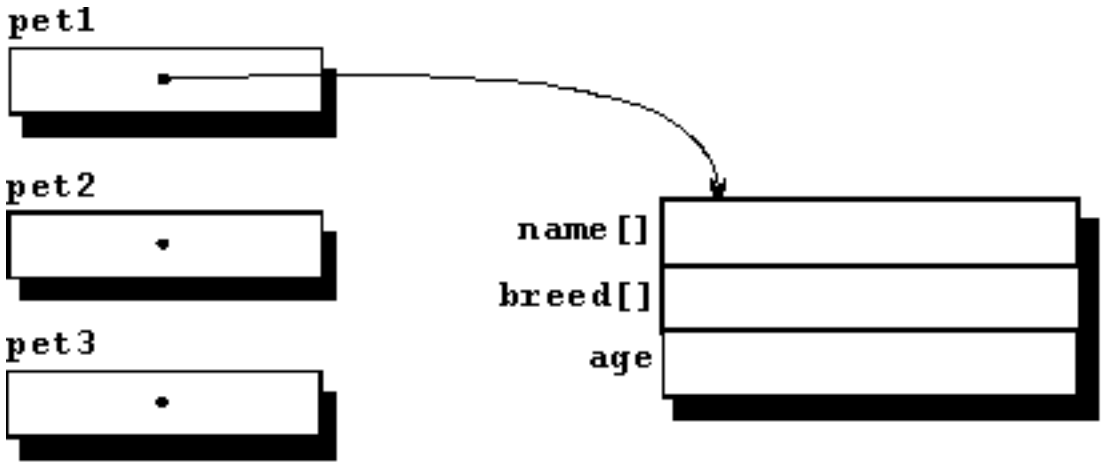

Figure 12-1

USING THE DYNAMICALLY ALLOCATED STRUCTURE

If you remember our studies of structures and pointers, you will recall that if we have a structure with a pointer pointing to it, we can access any of the variables within the structure. In lines 20 through 22 of the program, we assign some silly data to the structure for illustration. It should come as no surprise to you that these assignment statements look just like assignments to statically defined variables. Figure 12-2 illustrates the state of the data space following execution of line 22.

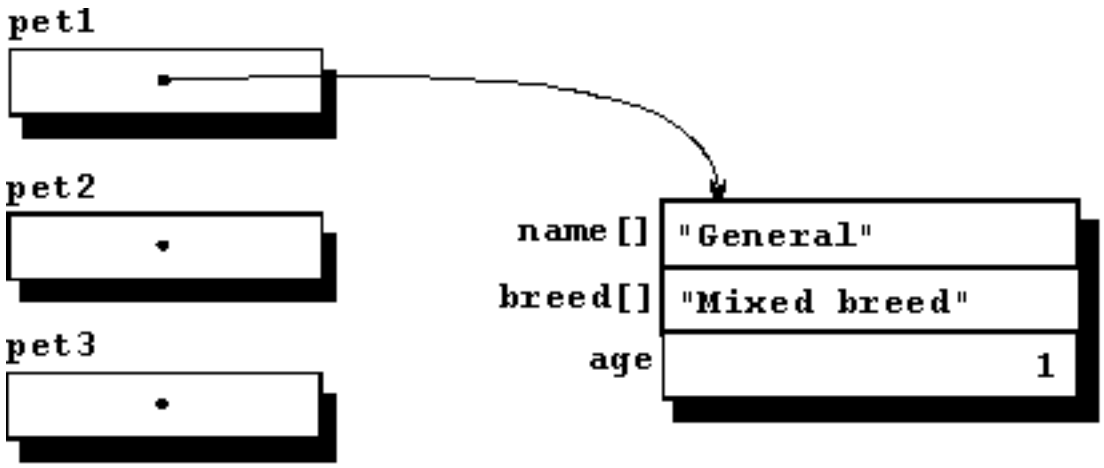

Figure 12-2

In line 24, we assign the value of **pet1** to **pet2** also via a pointer assignment statement which we introduced in chapter 8. This creates no new data, we simply have two pointers to the same object. Since **pet2** is pointing to the structure we created above, **pet1** can be reused to get another dynamically allocated structure which is just what we do next. Keep in mind that **pet2** could have just as easily been used for the new allocation. The new structure is filled with silly data for illustration in lines 27 through 29.

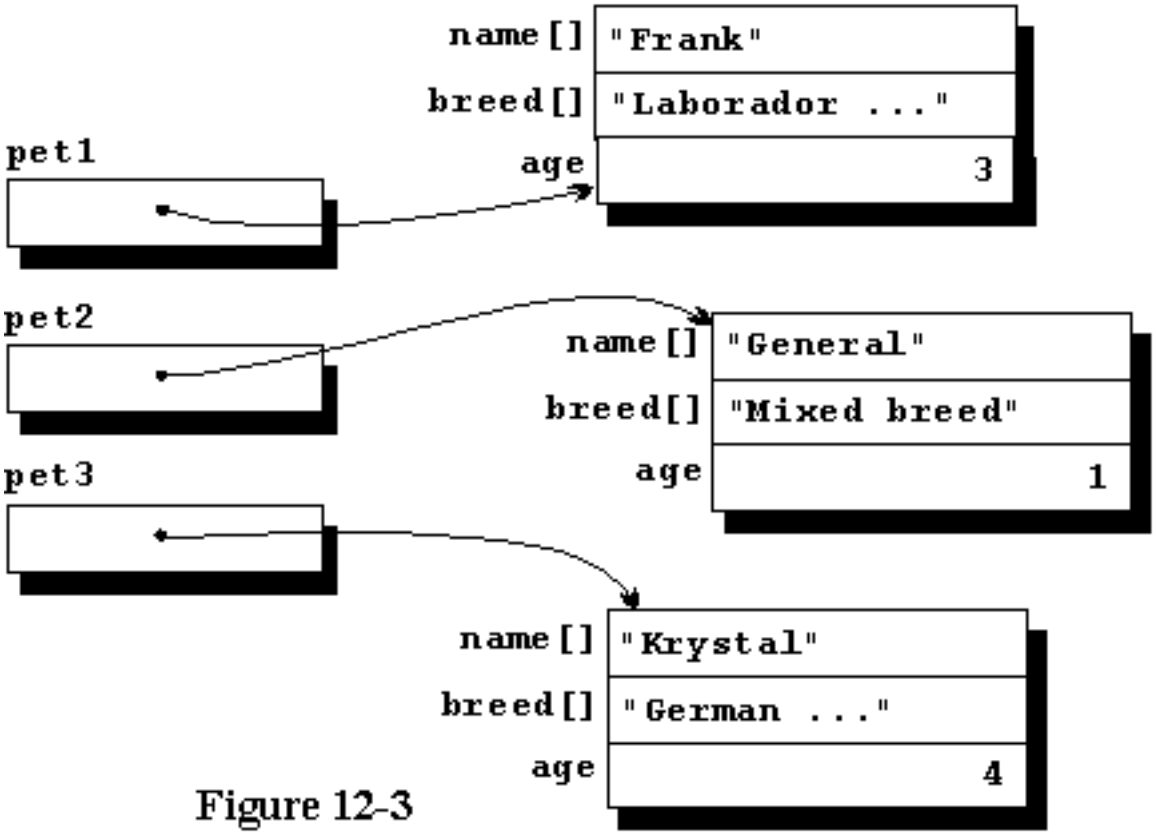

Figure 12-3

Finally, we allocate another block on the heap using the pointer **pet3**, and fill its block with illustrative data. Figure 12-3 illustrates the condition of the data space after execution of line 34 of the program.

Printing the data out should pose no problem to you since there is nothing new in the three print statements.

Even though it is not illustrated in this tutorial, you can dynamically allocate and use simple variables such as a single **char** type variable. This should be discouraged however since it is very inefficient.

GETTING RID OF THE DYNAMICALLY ALLOCATED DATA

Another new function is used to get rid of the data and free up the space on the heap for reuse, the function **free()**. To use it, you simply call it with the pointer to the dynamically allocated block of data as the only argument, and the block is deallocated.

In order to illustrate another aspect of the dynamic allocation and deallocation of data, an additional step is included in the program on your monitor. The pointer **pet1** is assigned the value of **pet3** in line 47. In doing this, the block that **pet1** was pointing to is effectively lost since there is no pointer that is now pointing to that block. It can therefore never again be referred to, changed, or deallocated. That memory, which is a block on the heap, is wasted from this point on. This is not something that you would ever purposely do in a program. It is only done here for illustration.

The first **free()** function call removes the block of data that **pet1** and **pet3** were pointing to, which deallocates the structure and returns the memory to the heap for further use. The second **free()** call deallocates the block of data that **pet2** was pointing to. We therefore have lost access to all of our data generated earlier. There is still one block of data that is on the heap but there is no pointer to it since we lost the address to it. Figure 12-4 illustrates the data space as it now appears. Trying to free the data pointed to by **pet1** would result in an error because it has already been freed by the use of **pet3**. There is no need to worry, however. When we return to the operating system, the entire heap will be disposed of with no regard to what we have put on it. The point does need to made that, if you lose a pointer to a block of the heap, it forever removes that block of data storage from our use and we may need that storage later.

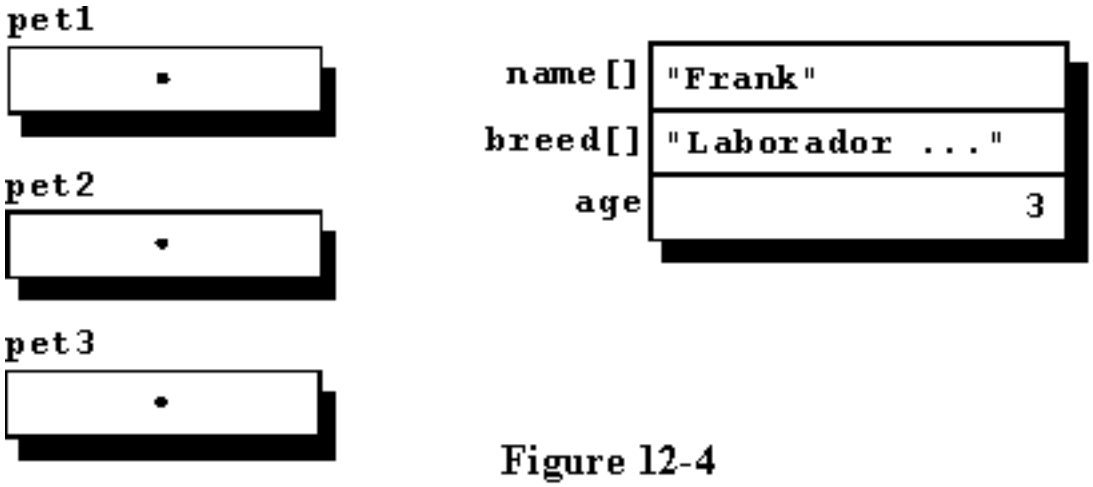

Figure 12-4

Compile and run the program to see if it does what you think it should do based on this discussion.

THAT WAS A LOT OF DISCUSSION

It took several pages to get through the discussion of the last program but it was time well spent. It should be somewhat exciting to you to know that there is nothing else to learn about dynamic allocation, the previous discussion covered it all. Of course, there is a lot to learn about the technique of using dynamic allocation, and for that reason, there are two more example programs to study.

AN ARRAY OF POINTERS

Example program ------> **BIGDYNL.C**

Load and display the file BIGDYNL.C for another example of dynamic allocation. This program is very similar to the last one since we use the same structure, but this time we define an array of pointers to illustrate the means by which you could build a large database using an array of pointers rather than a single pointer to each element. To keep it simple we define 12 elements in the array and another working pointer named **point**.

The ***pet[12]** is new to you so a few words would be in order. What we have defined is an array of 12 pointers, the first being **pet[0]**, and the last **pet[11]**. Actually, since an array is itself a pointer, the name **pet** by itself is a constant pointer to a pointer. This is valid in C, and in fact you can go farther if needed but you will get quickly confused. There is no limit as to how many levels of pointing are possible, so the definition **int ****pt**; is legal as a pointer to a pointer to a pointer to a pointer to an integer type variable, if I counted right. Such usage is discouraged until you gain considerable experience.

Now that we have 12 pointers which can be used like any other pointer, it is a simple matter to write a loop to allocate a data block dynamically for each and to fill the respective fields with any data desirable.

In this case, the fields are filled with simple data for illustrative purposes, but we could be reading from a database, from some test equipment, or from any other source of data.

You will note that this time we carefully check the return value from the **malloc()** function to see that it contains a non-zero value. If it returns a NULL value, we print a message that the allocation failed and exit the program. In a real production program, you would not simply terminate the program. It would be much more agreeable with the user to report the error to him and give him the opportunity to make more memory available before actually terminating the program. Error handling and recovery is a topic you will need to study someday, but it is beyond the scope of your programming experience at this point.

A few fields are randomly picked in lines 33 through 35 to receive other data to illustrate that simple assignments can be used, and the data is printed out to the monitor. The pointer **point** is used in the printout loop only to serve as an illustration. The data could have been easily printed using the **pet[n]** means of definition. Finally, all 12 blocks of data are freed before terminating the program.

Compile and run this program to aid in understanding this technique. There was nothing new here about dynamic allocation, only about an array of pointers.

A LINKED LIST

Example program ------> **DYNLINK.C**

We finally come to the granddaddy of all programming techniques as far as being intimidating. Load the program DYNLINK.C for an example of a dynamically allocated linked list. It sounds terrible, but after a little time spent with it, you will see that it is simply another programming technique made up of simple components that can be a powerful tool.

In order to set your mind at ease, consider the linked list you used when you were a child. Your sister gave you your birthday present, and when you opened it, you found a note that said, "Look in the hall closet." You went to the hall closet, and found another note that said, "Look behind the TV set." Behind the TV you found another note that said, "Look under the coffee pot." You continued this search, and finally you found your pair of socks under the dogs feeding dish. What you actually did was to execute a linked list, the starting point being the wrapped present and the ending point being under the dogs feeding dish. The list ended at the dogs feeding dish since there were no more notes.

In the program DYNLINK.C, we will be doing the same thing your sister forced you to do. However, we will do it much faster and we will leave a little pile of data at each of the intermediate points along the way. We will also have the capability to return to the beginning and traverse the entire list again and again if we so desire.

THE DATA DEFINITIONS

This program starts similarly to the last two with the addition of a constant declaration to be used later. The structure is nearly the same as that used in the last two programs except for the addition of another field within the structure in line 13, the pointer. This pointer is a pointer to another structure of this same type and will be used to point to the next structure in order. To continue the above analogy, this pointer will point to the next note, which in turn will contain a pointer to the next note after that.

We define three pointers to this structure for use in the program, and one integer to be used as a counter, and we are ready to begin using the defined structure for whatever purpose we desire. In this case, we

will once again generate nonsense data for illustrative purposes.

THE FIRST FIELD

Using the **malloc()** function, we request a block of storage on the heap and fill it with data being careful to check the return to assure that we did get a good allocation. The additional field in this example, the pointer, is assigned the value of NULL, which is used to indicate that this is the end of the list. We will leave the pointer named **start** pointing at this structure, so that it will always point to the first structure of the list. We also assign **prior** the value of **start** for reasons we will see soon. Keep in mind that the end points of a linked list will always have to be handled differently than those in the middle of a list. We have a single element of our list now and it is filled with representative data. Figure 12-5 is the graphical representation of the data space following execution of line 33.

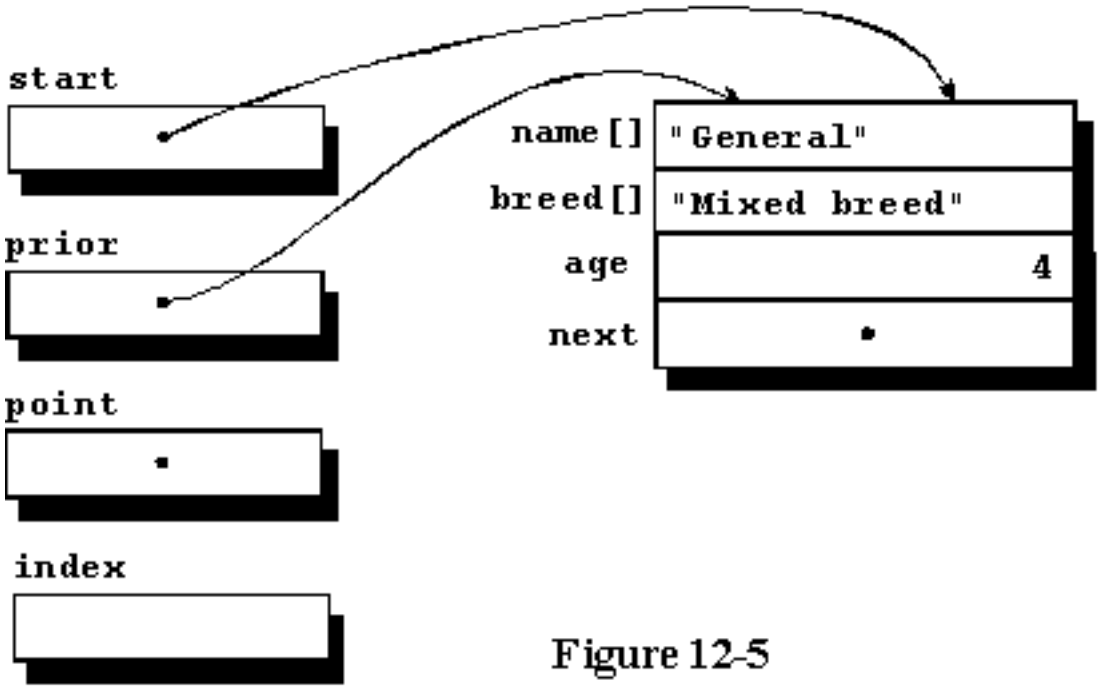

Figure 12-5

FILLING ADDITIONAL STRUCTURES

The next group of assignments and control statements are included within a **for** loop so we can build our list fast once it is defined. We will go through the loop a number of times equal to the constant RECORDS defined at the beginning of the program. Each time we go through the loop, we allocate memory, test the allocation return, fill the first three fields with nonsense, and fill the pointers. The pointer in the last record is given the address of this new record because the **prior** pointer is pointing to the prior record. Thus **prior->next** is given the address of the new record we have just filled. The pointer in the new record is assigned the value NULL, and the pointer **prior** is given the address of this new record because the next time we create a record, this one will be the prior one at that time. That may sound confusing but it really does make sense if you spend some time studying it.

Figure 12-6 illustrates the data space following execution of the loop two times. The list is growing downward by one element each time we execute the statements in the loop. When we have gone through the **for** loop 6 times, we will have a list of 7 structures including the one we generated prior to the loop. The list will have the following characteristics.

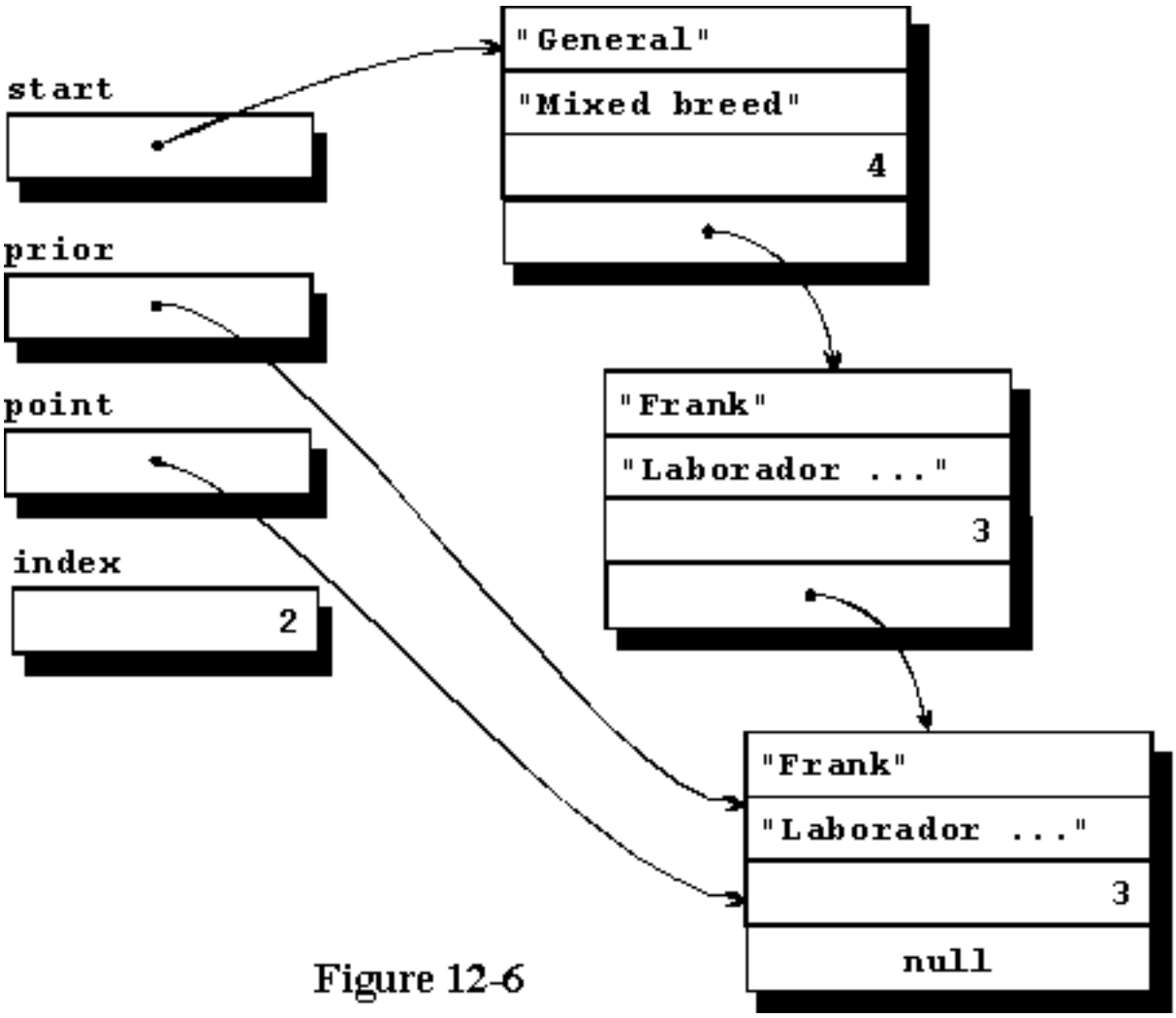

Figure 12-6

1. The pointer named **start** points to the first structure in the list.
2. Each structure contains a pointer to the next structure.
3. The last structure has a pointer containing the value NULL which can be used to detect the end.

It should be clear to you, if you understand the overall data structure, that it is not possible to simply jump into the middle of the list and change a few values. The only way to get to the third structure is by starting at the beginning and working your way down through the list one record at a time. Although this may seem like a large price to pay for the convenience of putting so much data outside of the program area, it is actually a very good way to store some kinds of data.

A word processor would be a good application for this type of data structure because you would never need to have random access to the data. In actual practice, this is the basic type of storage used for the text in a word processor with one line of text per record. Actually, a program with any degree of sophistication would use a doubly linked list. This would be a list with two pointers per record, one pointing down to the next record, and the other pointing up to the record just prior to the one in question. Using this kind of a record structure would allow traversing the data in either direction.

PRINTING THE DATA OUT

To print the data out, a similar method is used as that used to generate the data. The pointers are initialized and are then used to go from record to record, reading and displaying each record one at a time. Printing is terminated when the NULL in the last record is found, so the program doesn't even need to know how many records are in the list. Finally, the entire list is deleted to make room in memory for any additional data that may be needed, in this case, none. Care must be taken to assure that the last record is not deleted before the NULL is checked. Once the data is gone, it is impossible to know if you are finished yet.

MORE ABOUT DYNAMIC ALLOCATION AND LINKED LISTS

It is not difficult, nor is it trivial, to add elements into the middle of a linked list. It is necessary to create the new record, fill it with data, and point its pointer to the record it is desired to precede. If the new record is to be installed between the 3rd and 4th, for example, it is necessary for the new record to point to the 4th record, and the pointer in the 3rd record must point to the new one. Adding a new record to the beginning or end of a list are each special cases. Consider what must be done to add a new record in a doubly linked list.

Entire books are written describing different types of linked lists and how to use them, so no further detail will be given. The amount of detail given should be sufficient for a beginning understanding of C and its capabilities.

TWO MORE FUNCTIONS

Two additional functions must be mentioned, the **calloc()** and the **realloc()** functions. The **calloc()** function allocates a block of memory and clears it to all zeros which may be useful in some circumstances. It is similar to **malloc()** and will be left as an exercise for you to read about and use **calloc()** if you desire. Generally, you allocate a block of memory and immediately fill it with meaningful data so it wastes time to fill it with zeros in the **calloc()**, then fill it with real data. For this reason, the **calloc()** function is rarely used.

The **realloc()** is used to change the size of an allocated block, either bigger or smaller. You should ignore this until you gain a lot of experience with C. It is rarely used, even by experienced programmers.

PROGRAMMING EXERCISES

1. Rewrite the example program STRUCT1.C from chapter 11 to dynamically allocate the two structures.
2. Rewrite the example program STRUCT2.C from chapter 11 to dynamically allocate the 12 structures.

Return to Table of Contents

Advance to Chapter 13

C Tutorial - Chapter 13

CHARACTER AND BIT MANIPULATION

UPPER AND LOWER CASE

Example program ------> **UPLOW.C**

Examine the program named UPLOW.C for an example of a program that does lots of character manipulation. More specifically, it changes the case of alphabetic characters. It illustrates the use of four functions that have to do with case. It should be no problem for you to study this program on your own and understand how it works. The four functions on display in this program are all within the user written function, **mix_up_the_chars()**. Compile and run the program with the file of your choice. The four functions are;

```
isupper(c);    Is the character upper case?
islower(c);    Is the character lower case?
toupper(c);    Make the character upper case.
tolower(c);    Make the character lower case.
```

Many more classification and conversion routines are listed in the reference material for your compiler. You should spend time studying these at this time to get an idea of what functions are available.

CLASSIFICATION OF CHARACTERS

Example program ------> **CHARCLAS.C**

Load and display the next program, CHARCLAS.C for an example of character counting. We have repeatedly used the backslash n character representing a new line. These are called escape sequences, and some of the more commonly used are defined in the following table;

```
\n    Newline
\t    Tab
\"    Double quote
\\    Backslash
\0    NULL (zero)
```

Consult your compiler documentation for a complete list of escape sequences available with your compiler.

By preceding each of the above characters with the backslash character, the character can be included in a line of text for display, or printing. In the same way that it is perfectly all right to use the letter n in a line of text as a part of someone's name, and as an end-of-line, the other characters can be used as parts of text or for their particular functions.

This example program uses three of the functions that can determine the class of a character, and counts the characters in each class. The number of each class is displayed along with the line itself. The three functions are as follows;

```
isalpha(c); Is the character alphabetic?
```

```
isdigit(c); Is the character a numeral?
isspace(c); Is the character any of, \n, \t, or blank?
```

As noted above, many more classification routines are available with your compiler.

This program should be simple for you to find your way through, so no explanation will be given. It was necessary to give an example with these functions used. Compile and run this program with any file you choose.

THE LOGICAL FUNCTIONS

Example program ------> **BITOPS.C**

Load and display the program BITOPS.C. The functions in this group of functions are used to do bitwise operations, meaning that the operations are performed on the bits as though they were individual bits. No carry from bit to bit is performed as would be done with a binary addition. Even though the operations are performed on a single bit basis, an entire byte or integer variable can be operated on in one instruction. The operators and the operations they perform are given in the following table;

```
&   Logical AND, if both bits are 1, the result is 1.
|   Logical OR, if either bit is one, the result is 1.
^   Logical XOR, (exclusive OR), if one and only one
                          bit is 1, the result is 1.
~   Logical invert, if bit is 1, the result is 0, and
                        if bit is 0, the result is 1.
```

The example program uses several fields that are combined in each of the ways given above. The data is in hexadecimal format. It will be assumed that you already know hexadecimal format if you need to use these operations. If you don't, you will need to study it on your own. Teaching the hexadecimal format of numbers is beyond the scope of this tutorial. Be sure to compile and execute this program and observe the output.

THE SHIFT INSTRUCTIONS

Example program ------> **SHIFTER.C**

The last two operations to be covered in this chapter are the left shift and the right shift instructions. Load the example program SHIFTER.C for an example using these two instructions. The two operations use the following operators;

```
<<    n Left shift n places.
>>    n Right shift n places.
```

Once again the operations are carried out and displayed using the hexadecimal format. The program should be simple for you to understand on your own, there is no tricky code.

WHERE DO I GO FROM HERE?

Now that you have completed this tutorial, you are filled with knowledge of the C programming language, but you have relatively little experience with using it. I can make three recommendations to improve your knowledge of C and to give you additional exposure to it.

First, obtain a copy of the second edition of "The C Programming Language" written by Brian Kernighan and Dennis Ritchie, Prentice Hall, 1988. A careful reading of this book will provide you with a wealth of knowledge of the C programming language including many details which I felt were beyond the scope of this beginning tutorial. The book does not cover prototyping, since it was added later by the ANSI-C standardization committee, but that is the only major deficiency in the book and is easily compensated for. Simply use prototypes and the modern method of function definition when studying any of the example programs.

Secondly, and probably the most important recommendation, is to write programs in C. Writing C code, finding and fixing the errors that you will inadvertently introduce into the code, and finally seeing your program execute just the way you intended it to, provides a great feeling of accomplishment.

The third recommendation is to read current information on the language. Good sources for information are programming magazines, possibly a new book on C, or time spent reading Usenet newsgroups such as comp.lang.c or comp.lang.c.moderated. There are other newsgroups devoted to various operating systems, or to specific compilers which you may find interesting and informative.

The more you expose yourself to the C programming language, the more you will learn about it, and the more you will enjoy using it.

Good luck!

Return to Table of Contents

Gordon Dodrill - dodrill@swcp.com - Please email any comments or suggestions.