C Language

By Pythonlife

Introduction

The **C** programming language 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 was a great interest in running it on the IBM PC and compatibles, and other systems. C is excellent for actually writing system level programs, and the entire Applix 1616/OS operating system is written in C (except for a few assembler routines). 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.

It is not a good "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. It is very easy to write a program that destroys itself due to the silly little errors that, say, a Pascal compiler will flag and call a fatal error. In C, you are very much on your own, as you will soon find.

Since C is not a beginners language, I will assume you are not a beginning programmer, and I will not attempt to bore you by defining a constant and a variable. You will be expected to know these basic concepts. You will, however, not be expected to know anything of the C programming language. I will begin with 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 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 50 to 100% increase in runtime, because no language is as compact or 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.

Approximately 75 percent of all new commercial programs introduced for the IBM PC have been written in C, and the percentage is probably growing. Apple Macintosh system software was formerly written in Pascal, but is now almost always written in C. The entire Applix 1616 operating system is written in C, with some assembler routines.

Since C was designed essentially by one person, and not by a committee, it is a very usable language but not too closely defined. There was no official standard for the C language, but the American National Standards Association (ANSI) has developed a standard for the language, so it will follow rigid rules. It is interesting to note, however, that even though it did not havea standard, the differences between implementations are usually small. This is probably due to the fact that the original unofficial definition was so well thought out and carefully planned that extensions to the language are not needed.

Even though the C language enjoys a good record when programs are transported from one implementation to another, there are differences in compilers, as you will find any time you try to use another compiler. Most of the differences become apparent when you use nonstandard extensions such as calls to the MS-DOS BIOS, or the Applix 1616/OS system calls, but even these differences can be minimized by careful choice of programming means.

Applix 1616 builders have only the HiTech C compiler available. This version of the tutorialis customised to suit HiTech C. The original MS-DOS version by Gordon Dodrill was portedto the Applix 1616 (with great effort) by Tim Ward, and typed up by Karen Ward. The programs have been converted to HiTech C by Tim Ward and Mark Harvey, while Kathy Morton assisted greatly in getting Visual Calculator working. All have been tested on the Applix 1616/OS multitasking operating system. The Applix distribution disks contain the complete original text of this tutorial, plus all the converted C source code. The second disk contains executable, relocatable versions of all the programs, ready to run on an Applix 1616. There is also a directory of the original IBM source code, for those using IBM computers, who may wish to try them with a different compiler. This printed version has been edited, indexed and pretty printed by Eric Lindsay, who added the Applix specific material.

This printed version of the tutorial includes copies of all the code, for easier reference. It also includes a comprehensive table of contents, and index.

1
Getting Started

This tutorial can be read simply as a text, however it is intended to be interactive. That is, you should be compiling, modifying and using the programs that are presented herein.

All the programs have been tested using the HiTech C compiler, and we assume that you have a copy of this. In addition, you should have a copy of various updates and header files for the C compiler, which appear on Applix User disks.

You can use either the builtin Applix 1616/OS editor `edit`, or the $30 *Dr Doc* editor in non-document mode. *Dr Doc* is somewhat more powerful, however as it loads from disk, it is slightly slower to get started. The source code has been edited to suit a tab setting of 5, so invoke your editor with tabs set to a spacing of 5. For example, `edit sourcecode.c 5` would let you edit a file called `sourcecode.c`.

Before you can really use C, there are certain equipment requirements that must be met. You must have a disk co-processor card, and at least one disk drive. If your drives are smaller than 800k, you will probably require two disk drives. We assume you either have 1616/OS Version4 multitasking, or else have an `assign` MRD available on your boot disk.

You should make use of the `xpath`, and the `assign` commands to set up your boot disk in a form suitable for use with C. This should be done in the `autoexec.shell` file on your boot disk, as set out below.

1.1 C Boot Disk

Make a new, bootable copy of your 1616 User disk, following the directions in your *Users Manual*. To ensure sufficient space, delete any obviously unwanted files you notice on the copy.

Copy the contents of your HiTech C distribution disk to the new disk, keeping the subdirectories the same as on the HiTech disk.

If you have received any updated C header files or other updates, copy these also to their respective subdirectories on your new disk.

Using `edit`, alter the `xpath` and `assign` commands in your `autoexec.shell` file in the root directory of your new disk.

Your `xpath` should include `/F0/bin` (if it is not already included).

Add the following lines to your `autoexec.shell`, to recreate the environment used by Tim Ward when originally running these programs.

```
assign /hitech /f0/bin
assign /sys /f0/include
assign /temp /rd
```

This will allow code to be written without regard to where you actually put your files. If you are using a second drive, or a hard disk, simply change the assign to point `/hitech` to the correct drive. C tends to use temporary files extensively. If you have sufficient memory available on your ram disk, use `/rd` for temporary files. If not, use the current drive and directory, as indicated by the `assign /temp .`

Make sure you copy the new C preprocessor `relcc.xrel` from the user disk into the `/bin` subdirectory of your new C disk.

Note that `relcc` expects by default to find its C library files on the current drive in the `/hitech` directory. It also expects to find its include files on the current drive in the `/hitech/include` directory. We will explain what this means later, and there is a detailed discussion of the HiTech C compiler at the end of the tutorial.

If all is correct, you can now compile a C file by typing

```
relcc -v file.c
```

The `-v` flag is to invoke the verbose mode, which produces the maximum information from the compiler.

If you are experimenting, you may prefer to capture any errors encountered in a file, for later study. If so, use

```
relcc -v file.c } errorfile
```

1.2 What Is An Identifier?

Before you can do anything in any language, you must at least know how you name an identifier. An indentifier is used for any variable, function, data definition, etc. In the programming language C, 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 is not the same as using "index" and neither of them is the same as using "InDex" for a variable. All three refer to different variables.

2. As C is defined, up to eight significant characters can be used and will be considered significant. If more than eight are used, they may be ignored by the compiler. This may or may not be true of your compiler. You should check your reference manual to find out how many characters are significant for your compiler. The HiTech C compiler used with the Applix 1616 allows 31 significant characters, and prepends an underscore (_)

It should be pointed out that some C compilers allow use of a dollar sign in an identifier name, but since it is not universal, it will not be used anywhere in this tutorial. Check your documentation to see if it is permissible for your particular compiler.

1.3 What About The Underline?

Even though the underline can be used as part of a variable name, it seems to be used very little by experienced C programmers. 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 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 this reason.

1.4 How This Tutorial Is Written

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 this 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 the program coding, time spent planning the data structures will be well spent and the final program will reflect the original planning.

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. This method, however, leads to a lack of knowledge in how the various parts are combined. For that reason, the last chapter is devoted entirely to using the features taught in the earlier chapters. It will illustrate how to put the various features together to create a usable program. They are given for your study, and are not completely explained. Enough details of their operation are given to allow you to understand how they work after you have completed all of the previous lessons.

1.5 A Discussion Of Some Of The Files

Many of the files in this tutorial are unduely IBM specific. Details of the Applix 1616 versions of these normally supplement these notes, although some discussion of MS-DOS features still remain.

1.6 List.xrel

This file will list the source files for you with line numbers and filename. To use it, simply type "LIST" followed by the appropriate filename. Type `list firstex.c` now for an example. The C source code is given later in Chapter 14 along with a brief description of its operation. Applix 1616 users always have the inbuilt `edit` comand available to them, so this program isn't really essential.

2
Getting started in C

The best way to get started with C is to actually look at a program, so load the file named `trivial.c` into `edit` and display it on the monitor.

2.1 Your First C Program

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.

```
main()
{
}
```

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 **{ }**, 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.

2.2 A Program That Does Something

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.

```
main( )
{
printf("This is a line of text to output.");
}
```

The executable statement is another function. Once again, we will not worry about what a function is, but only how to use this one. In order to output text to the monitor, it is put within the function parentheses and bounded by quotation marks. The end result is that whatever 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 the line. 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 programis also executable, so you can compile and run it to see if it does what you think it should. With some compilers, you may get an error message while compiling, indicating the printf() should have been declared as an integer. Ignore this for the moment.

2.3 Another Program With More Output

Load the program `wrtmore.c` and display it on your monitor for an example of more output and another small but important concept. You will see that there are four program statements in this program, each one being a "printf" function statement. The top line will be executed first then the next, and so on, until the fourth line is complete. The statements are executed in orderfrom top to bottom.

```
main( )
{
   printf("This is a line of text to output.\n");
   printf("And this is another ");
   printf("line of text.\n\n");
   printf("This is the third line.\n");
}
```

Notice the funny character near the end of the first line, namely the backslash. The backslash is used in the printf statement to indicate 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. It is commonly referred to as a carriage return/line feed. Any place within text that you desire, you can put a newline character and start a new line. You could even put it in the middle of a word and split the word between two lines. The C compiler considers the combination of the backslash and letter n as one character. The exact characters used to indicate a newlin and carriage return are operating system specific. MS-DOS, Unix, 1616/OS and Macintosh may vary one from the other.

A complete description of this program is now possible. The first printf outputs a line of text and returns the carriage. The second printf outputs a line but does not return the carriage so the third line is appended to that 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.

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

2.4 To Print Some Numbers

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.

```
main( )
{
int index;
   index = 13;
   printf("The value of the index is %d\n",index);
   index = 27;
   printf("The valve of the index = %d\n",index);
   index = 10;
   printf("The value of the index = %d\n",index);
}
```

The entry point "main" should be clear to you by now as well as the beginning brace. The first new thing we encounter is the line containing "int index;", which is used to define an integer variable named "index". The "int" is a reserved word in C, and can therefore not be used for anything else. It defines a variable that can have a value from -32768 to 32767 on most MS-DOS microcomputer implementations of C. It defines a variable with a value from -2147483648 to 2147483647 in HiTech C. Consult your compiler users manual for the exact definition for your compiler. The variable name, "index", can be any name that follows the rules for an identifier

and is not one of the reserved words for C. Consult your manual for an exact definition of an identifier for your compiler. In HiTech C, the construction of identifier names is the same as in UNIX, however 31 characters and both cases are significant. The compiler prepends an underscore to external references in the assembler pass. The final character on the line, the semi-colon, is the statement terminator used in C.

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 first one assigns the value of 13 to "index", and its value is printed out. (We will see how shortly.) Later, the value 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. 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.

2.5 How Do We Print Numbers

To keep our promises, 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, namely output a variable. The % sign is used to signal the start 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, and after the 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. We could add more "%d" output field descriptors within the brackets and more variables following the description to cause more data to be printed with one statement. Keep in mind however, that it is important that the number of field descriptors and the number of variable definitions must be the same or the runtime system will get confused and probably quit with a runtime error.

Much more will be covered at a later time on all aspects of input and output formatting. A reasonably good grasp of this topic is necessary in order 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.

2.6 How Do We Add Comments In C

Load the file `comments.c` and observe it on your monitor for an example of how comments can be added to a C program.

```
          /* This is a comment ignored by the compiler */
main( )   /* This is another comment ignored by the compiler  */
{
   printf("We are looking at how comments are "); /* A comment is
                                                     allowed to be
                                                     continued on
                                                     another line  */
```

```
   printf("used in C.\n");
}
   /* One more comment for effect */
```

Comments are added to make a program more readable to you but the compiler must ignore the comments. The slash star combination is used in C for comment delimiters. They 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 first slash star combination introduces the first comment and the star at the end of the first 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 next comment 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 and continue 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. The last comment is located following the completion of the program, illustrating that comments can go nearly anywhere in a C program.

Experiment with this program be 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 printed out.

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 programa year from now than one with few or no comments. You will very quickly develop your ownpersonal style of commenting.

Some compilers allow you to "nest" comments which can be very handy if you need to "comment out" a section of code during debugging. Check your compiler documentation for the availability of this feature with your particular compiler. Compile and run `comments.c` at this time.

2.7 Good Formatting Style

Load the file `goodform.c` and observe it on your monitor.

```
main()     /* Main program starts here  */
{
   printf("Good form ");
   printf          ("can aid in ");
   printf                  ("understanding a program.\n");
   printf("And bad form ");
   printf          ("can make a program ");
   printf                  ("unreadable.\n");
}
```

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 concerning how you format 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.

Now load and display the program `uglyform.c` and observe it.

```
main( )    /* Main program starts here */{printf("Good form ");printf
   ("can aid in ");printf(" understanding a program.\n")
;printf("And bad form ");printf("can make a program ");
printf("unreadable.\n");}
```

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 language. Be observant of styles as you see C programs in magazines, books, and other publications.

This should pretty well cover the basic concepts of programming in C, but as there are many other things to learn, we will forge ahead to additional program structure.

2.8 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
Program Control

3.1 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.

Load the program `while.c` and display it for an example of a while loop.

```
/* This is an example of a "while" loop */
main( )
{
int count;
   count = 0;
   while (count < 6) {
       printf("The value of count is %d\n",count);
       count = count + 1;
   }
}
```

We begin with a comment and the program name, then go on to define an integer variable "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 parentheses is true, all statements within the braces will be executed. In this case, since the variablecount is incremented byone every timethe statements areexecuted,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 probably be correct.

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 braces but can stand alone.

Compile and run this program.

3.2 The Do-While Loop

A variation of the while loop is illustrated in the program `dowhile.c,` which you should load and display.

```
/* This is an example of a do-while loop */
main( )
{
int i;
   i = 0;
   do {
       printf("the value of i is now %d\n",i);
```

```
        i = i + 1;
    } while (i < 5);
}
```

This program is nearly identical to the last one except that the loop begins with the reserved word "do", followed by a compound statement in braces, then the reserved word "while", and finally an expression in parentheses. The statements in the braces are executed repeatedly as long as the expression in 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 this statement. 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 "i", were not changed within the loop, the loop would never terminate, and hence the program would never terminate. Finally, just like the while loop, if only one statement will be executed within the loop, no braces are required. Compile and run this program to see if it does what you think it should do.

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.

3.3 The For Loop

The "for" loop is really nothing new, it is simply a new way of describe the "while" loop. Load and edit the file named `forloop.c` for an example of a program with a "for" loop.

```
/* This is an example of a for loop */
main( )
{
int index;
    for(index = 0;index < 6;index = index + 1)
        printf("The value of the index is %d\n",index);
}
```

The "for" loop consists of the reserved word "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 loop through the program. 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 executed 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.

3.4 The If Statement

Load and display the file `ifelse.c` for an example of our first conditional branching statement, the "if".

```
/* This is an example of the if and if-else statements */
main()
{
int data;
   for(data = 0;data < 10;data = data + 1) {
       if (data == 2)
           printf("Data is now equal to %d\n",data);
       if (data < 5)
           printf("Data is now %d, which is less than 5\n",data);
       else
           printf("Data is now %d, which is greater than 4\n",data);
   }   /* end of for loop     */
}
```

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 statement 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. 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.)

3.5 Now For The If-Else

The second "if" is similar to the first, with the addition of a new reserved word, the "else", following the first printf statement. 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.

3.6 The Break And Continue

Load the file named `breakcon.c` for an example of two new statements.

```
main( )
{
int xx;
   for(xx = 5;xx < 15;xx = xx + 1){
       if (xx == 8)
           break;
       printf("in the break loop, xx is now %d\n",xx);
   }
   for(xx = 5;xx < 15;xx = xx + 1){
       if (xx == 8)
           continue;
       printf("In the continue loop, xx is the now %d\n",xx);
   }
}
```

Notice that in the first "for" 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 the statements 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 8, the loop is terminated and the last value printed will be the previous value, namely 7.

The next "for" loop, 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. Compile and run the program to see if it does what you expect.

3.7 The Switch Statement

Load and display the file `switch.c` for an example of the biggest construct yet in the C language, the switch.

```
main( )
{
int truck;
   for (truck = 3;truck < 13;truck = truck + 1) {
       switch (truck) {
           case 3  : printf("The value is three\n");
                     break;
           case 4  : printf("The value is four\n");
                     break;
           case 5  :
           case 6  :
           case 7  :
           case 8  : printf("The value is between 5 and 8\n");
                     break;
           case 11 : printf("The value is eleven\n");
                     break;
           default : printf("It is one of the undefined values\n");
                     break;
       } /* end of switch     */
   } /* end of the loop  */
}
```

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 desired are then enclosed within a pair of braces. The reserved word "case" is used to begin each case entered followed by the value of the variable, then a colon, and the statements to be executed.

In this example, if the variable "truck" contains the value 8 during this pass of the switch statement, the printf will cause "The value is three" to be displayed, and the "break" statement will cause us to jump out of the switch.

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 has the value 5, the statements will begin executing 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 the "case 8" portion will direct the execution out the bottom of the switch. 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.

Compile and run `switch.c` to see if it does what you expect it to after this discussion.

3.8 The Goto Statement

Load and display the file `gotoex.c` for an example of a file with some "goto" statements in it.

```
main()
{
int dog,cat,pig;
   goto real_start;
   some_where:
   printf("This is another line of the mess.\n");
   goto stop_it;
/* the following section is the only section with a useable goto */
   real_start:
   for(dog = 1;dog < 6;dog++) {
       for(cat = 1;cat < 6;cat++) {
           for(pig = 1;pig < 4;pig++) {
               printf("Dog = %d  Cat = %d  Pig = %d\n",dog,cat,pig);
               if ((dog + cat + pig) > 8 ) goto enough;
           };
       };
   };
   enough: printf("Those are enough animals for now.\n");
/* this is the end of the section with a useable goto statement    */
   printf("\nThis is the first line out of the spaghetti code.\n");
   goto there;
   where:
   printf("This is the third line of the spaghetti code.\n");
   goto some_where;
   there:
   printf("this is the second line of the spaghetti code.\n");
   goto where;
   stop_it:
   printf("This is the last line of the mess.\n");
}
```

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 are not allowed to jump into any loop, but you are allowed to jump out of a loop. Also, you are not allowed to jump out of any function into another. These attempts will be flagged by your compiler as an error if you attempt any of them.

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 17 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 all three loops but one "goto" statement gets you out of all three.

Some persons say the "goto" statement should never be used under any circumstances but this is rather narrow minded thinking. If there is a place where a "goto" will be best, 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. These will be left to your study. One point of reference is the Visual Calculator described in Chapter 14 of this tutorial. This program is contained in four separately compiled programs and is a rather large complex program. If you spend some time studying the source code, you will find that there is not a single "goto" statement anywhere in it. 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.

3.9 Finally, A Meaningful Program

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.

```
/*********************************************************/
/*     This is a temperature conversion program written in  */
/*     the C programming language. This program generates   */
/*     and displays a table of farenheit and centigrade     */
/*     temperatures, and lists the freezing and boiling     */
/*     of water                                             */
/*********************************************************/
main( )
{
int count;            /* a loop control variable           */
int farenheit;        /* the temperature in farenheit degrees */
int centigrade;   /* the temperature in centigrade degrees */
   printf("Centigrade to farenheit temperature table\n\n");
   for(count = -2;count <= 12;count = count + 1 ){
       centigrade = 10 * count;
       farenheit = 32 + (centigrade * 9)/5;
   printf("  C =%4d   F =%4d  ",centigrade,farenheit);
   if (centigrade == 0)
       printf(" Freezing point of water");
   if (centigrade == 100)
       printf(" Boiling point of water");
   printf("\n");
  }  /* end of for loop */
}
```

Of particular importance is the formatting. The header is simply several lines of comments describing what the program does in a manner the 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 a few spaces to the right of the "for" reserved word, and the closing brace is lined up under the "f" in "for". This makes debugging a bit easier because the construction becomes very obvious.

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 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 opportunity to define each with a comment.

3.10 Another Poor Programming Example

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 actually 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.

```
main( )
{
int x1,x2,x3;
   printf("Centigrade to Farenheit temperature table\n\n");
   for(x1 = -2;x1 <= 12;x1 = 1){
       x3 = 10 * x1;
       x2 = 32 + (x3 * 9)/5;
       printf("  C =%4d   F =%4d  ",x3,x2);
       if (x3 == 0)
           printf("Freezing point of water");
```

```
        if (x3 == 100)
            printf("Boiling point of water");
        printf("\n");
    }
}
```

Compile and run this program to see that it does exactly what the last one did.

3.11 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.

4
Assignment & Logical compares

4.1 Integer Assignment Statements

Load the file `intasign.c` and display it for an example of assignment statements.

```
/* This program will illustrate the assignment statements */

main( )
{
int a,b,c;    /* Interger variables for examples    */

   a = 12;
   b = 3;
   c = a + b; /* simple addition     */
   c = a - b; /* simple subtraction */
   c = a * b; /* simple multiplication */
   c = a / b; /* simple addition     */
   c = a % b; /* simple modulo (remainder) */
   c = 12*a + b/2 - a*b*2/(a*c + b*2);
   c = c/4+13*(a + b)/3 - a*b + 2*a*a;
   a = a + 1; /* incrementing a variable    */
   b = b * 5;

   a = b = c = 20;    /* multiple assignment    */
   a = b = c = 12*13/4;
}
```

Three variables are defined for use in the program and the rest of the program is merely a series of illustrations of various assignments. The first two lines of the assignment statements assign numerical values to "a" and "b", and the next four 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 "int" or "char" type variables, and of course "int" extensions such as "long", "short", etc. Following these, there are two lines illustrating how to combine some of the variables in some 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.

The next two expressions 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.

This leaves us with the last two lines 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 it 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 last statement illustrates that it is possible to actually do some calculations to arrive at the value which will be assigned to all three variables.

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.

This would be a good time for a preliminary definition of a rule to be followed in C. The data definitions are always given before any executable statements in any program block. This is why the variables are defined first in this program and in any C program. If you try to define a new variable after executing some statements, the compiler will issue an error.

4.2 Additional Data Types

Loading and editing `mortypes.c` will illustrate how some additional data types can be used.

```
/* The purpose of the file is to introduce additional data types */
main( )
{
int a,b,c;            /* -32768 to 32767 with no decimal point */
char x,y,z;           /* 0 to 255 with no decimal point     */
float num,toy,thing;     /* 10E-38 to 10E+38 with decimal point     */
   a = b = c = -27;
   x = y = z = 'A';
   num = toy = thing = 3.6792;
   a = y;      /* a is now 65 (character A)   */
   x = b;      /* x will now be a funny number */
   num = b;       /* num will now be -27.00  */
   a = toy;       /* a will now be 3 */
}
```

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 values between zero and 255, since it is stored in only one byte of memory. 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 integer data type is stored in two bytes of computer memory on most 8 bit and MS-DOS based microcomputers.

The Applix 1616 uses a 68000 chip with true 32 bit registers, so integers are 32 bits. This means the range of an integer is not ±32767, but ±2147483648, or over two billion! On the Applix, a short int may be more appropriate in some places.

4.3 Data Type Mixing

It would be profitable at this time to discuss the way C handles the two types "char" and "int". Most functions in C that are designed to operate with integer type variables will work equally wellwith character type variables because they are a form of an integer variable. Those functions, 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 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, on most computers, has an allowable range of from 10^{-38} to 10^{+38}. Not all compilers have the same available range, so check your reference manual for the limits on your compiler.

4.4 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 "integer" data type, no special considerations need be taken to promote a "char" to an "int" variable. When going the other way, there is no standard, so you may simply get garbage if the value is within the range of zero to 255. In the second line therefore, when attempting to set x (a char) to -27, you may or may not get a well defined answer, it depends on your particular implementation of C.

The third line illustrates the simplicity of translating an integer into a "float", simply assign it the new value and the system will do the proper conversion. When going the other way 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.

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, but the next program will cover this for you.

4.5 Characteristics of Variable Types

One unfortunate problem with C (and some other languages) is that the actual size of variables is somewhat dependant upon the machine (and compiler) being used. Difficulties are sometimes experienced when moving code from one machine to another due to this.

A general rule of thumb for modern compilers is that `char` is at least 8 bits, `short` is at least 16 bits, `long` is at least 32 bits, `int` is the same as either short or long (which causes lots of code conversions come unstuck), `float` is at least 32 bits, and `double` is at least as wide as float. As a consequence, you should use either `short` or `long` in preference to `int` (despite its heavy use in this tutorial), and should avoid `double` where possible. Note that using `short` instead of `int` may save an extra memory access, and help speed up code, especially in large loops.

integer	32 bit (assembles as .l, or ±2147483648)
character	8 bit (0 to 255)
float	32 bit (about 10^{38})
double	64 bit (10^{308} maximum)
short	16 bit (assembles as .w, or ±32767)
long	32 bit (assembles as .l, or ±2147483648)

Byte ordering is another problem encountered when converting programs from one machine to another. In the 68000, the low byte of a 16 bit word is the second, odd numbered, byte. The bytes in a 32 bit long are laid out most significant byte at the address, with the least significant byte below it. Intel chips use the opposite order. This byte sex problem leads to arguments akin to those in religion and politics, and just as likely to lead to real solutions.

4.6 Lots Of Variable Types

Load the file `lottypes.c` and display it your screen.

```
main( )
{
int a;            /* simple integer type        */
long int b;       /* long integer type          */
short int c;      /* short integer type         */
unsigned int d;   /* unsigned integer type      */
char e;           /* character type             */
float f;          /* floating point type        */
double g;         /* double precision floating point */

   a = 1023;
   b = 2222;
   c = 123;
   d = 1234;
   e = 'X';
   f = 3.14159;
   g = 3.1415926535898;

   printf("a = %d\n",a);      /* decimal output      */
   printf("a = %o\n",a);      /* octal output        */
   printf("a = %x\n",a);      /* hexadecimal output */
   printf("b = %1d\n",b);        /* decimal long output     */
   printf("c = %d\n",c);      /* decimal short output   */
   printf("d = %u\n",d);      /* unsigned output        */
   printf("e = %c\n",e);      /* character output       */
   printf("f = %f\n",f);      /* floating output        */
   printf("g = %f\n",g);      /* double float output    */
   printf("\n");
   printf("a = %d\n",a);      /* simple int output      */
   printf("a = %7d\n",a);        /* use a field width of 7     */
   printf("a = %-7d\n",a);       /* left justify in field of 7 */
   printf("\n");
   printf("f = %f\n",f);      /* simple float output    */
   printf("f = %12f\n",f);       /* use field width of 12 */
   printf("f = %12.3f\n",f);  /* use 3 decimal places   */
   printf("f = %12.5f\n",f);  /* use 5 decimal places   */
   printf("f = %-12.5f\n",f);    /* left justify in field */
}
```

This file contains every standard simple data type available in the programming language C. There are other types, but they are the compound types that we will cover in due time.

Observe the file. First we define a simple "int" followed by a "long int" and a "short int". Consult your reference manual for an exact definition of these for your compiler, because they are not consistent from implementation to implementation. The "unsigned" is next and is defined as the same size as the "int" but with no sign. The "unsigned" then will cover a range of 0 to 65535 on MS-DOS microcomputers, but as 0 to 4294967296 in HiTech. It should be pointed out that when "long", "short", or "unsigned" is desired, the "int" is optional and left out by most experienced programmers. We have already covered the "char" and the "float", which leaves only the "double". The "double" usually 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 thanthesimple "float". Consult your referencemanual forthe rangeand accuracy of the"double".

Another diversion is in order at this point. Most compilers have provisions for floating point math, but only double floating math. They will promote a "float" to a "double" before doing calculations and therefore only one math library will be needed. Of course, this is totally transparent to you, so you don't need to worry about it. 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 requires 4 bytes of storage and a "double" requires 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. Your compiler may require a different number of bytes than 4 or 8. Consult your reference manual for the correct number of bytes used by your compiler.

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

4.7 The Conversion Characters

Following is a list of the conversion characters and the way they are used in the "printf" statement.

d	decimal notation
o	octal notation
x	hexadecimal notation
u	unsigned notation
c	character notation
s	string notation
f	floating point notation

(the inbuilt Applix ***printf*** function has a %e conversion character, that prints the internal error message corresponding to an error number - this is not standard in C.)

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

-	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 presently displayed on your monitor, with the exception of the string notation which will be covered later in this tutorial. 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 any of the fields anyway you desire.

4.8 Logical Compares

Load and view the file named `compares.c` for many examples of compare statements in C.

```
main( )  /* this file will illustrate logical compares */
{
int x = 11,y = 11,z = 11;
char a = 40,b = 40,c = 40;
float r = 12.987,s = 12.987,t = 12.987;
                  /* First group of compare statements */
   if (x == y) z = -13;   /* This will set z = -13  */
   if (x > z)  a = 'A';   /* This will set a = 65   */
   if (!(x > z)) a = 'B'; /* This will change nothing  */
   if (b <= c) r = 0.0;   /* This will set r = 0.0  */
   if (r != s) t = c/2;   /* This will set t = 20   */
                  /* Second group of compare statements */
   if (x = (r != s)) z = 1000; /* This will set x = some positive
                                  number and z = 1000 */
   if (x = y) z = 222;    /* This sets x = y, and z = 222   */
   if (x != 0) z = 333;   /* This sets z = 333 */
   if (x) z = 444;        /* This sets z = 444 */
                  /* Third group of compare statements  */
```

```
    x = y = z = 77;
    if ((x == y) && (x == 77)) z = 33; /* This sets z = 33  */
    if ((x >y) || (z > 12))    z = 22;  /* This sets z = 22  */
    if ((x && y && z) z = 11;           /* This sets z = 11  */
    if ((x = 1) && (y = 2) && (z = 3)) r = 12.00; /* This sets
                                  x = 1, y = 2, z = 3, r = 12 .00 */
    if ((x == 2) && (y = 3) && (z = 4)) r = 14.56; /* This doesn't
                                 change anything */
                    /* Fourth group of compares */
    if (x == x); z = 27.345;  /* z always gets changed  */
    if (x != x)  z = 27.345;  /* Nothing gets changed */
    if (x = 0)   z = 27.345;  /* This sets x = 0, z is unchanged */
}
```

We begin by defining and initializing nine variables to use in the following compare statements. This initialization is new to you and can be used to initialize variables while they are defined.

The first group of compare statements represents the simplest kinds of compares since they simply compare two variables. Either variable could be replaced with a constant and still be valid compare, but two variables is the general case. The first compare checks to see if "x" is equal to "y" and it uses the double equal sign for the comparison. 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 "x" is greater than "z".

The third introduces the "NOT" operator, the exclamation, which can be used to invert the resultof any logical compare. The fourth checks for "b" less than or equal to "c", and the last checksfor "r" not equal to "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 readability of the overall program.

4.9 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 a non-zero value. Any integer or char type of variable can be used for the result of a true/false test, or the result can be an implied integer or char.

Look at the first compare of the second group of compare statements. The expression "r is != s" will evaluate as a "true" since "r" was set to 0.0 above, so the result will be a non-zero value, probably 1. Even though the two variables that are compared are "float" variables, the result will be of type "integer". There is no explicit variable to which it will be assigned so the result of the compare is an implied integer. Finally the resulting number, 1 in this case, is assigned to the integer variable "x". If double equal signs were used, the phantom value, namely 1, wouldbe compared to the value of "x", but since the single equal sign is used, the value 1 is simply assigned to "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, probably 1, and we assigned "z" the value 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 ina conditional statement. The value assigned to "x" was probably a 1 but different compilers may assign a different value as long as it is non-zero.

The next example 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 therefore assigned 222.

The third example, in the second group, compares "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 is not actually needed and the result of the compare is true. The third and fourth examples of this group are therefore identical.

4.10 Additional Compare Concepts

The third group of compares will introduce some additional concepts, namely the logical "AND" and the logical "OR". 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". The entire statement reads, if "x' equals "y" AND if "x" equal 77 then the result is "true". Since this is true, the variable z is set equal to 33.

The next compare in this group introduces the "||" operator which is the "OR". 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" or not, because only one of the two conditions must be true for the result to be true. The result is true, so therefore "z" will be assigned the value of 22.

4.11 Logical Evaluation

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. Namely, 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.

4.12 Precedence Of Operators

The question will come up concerning the precedence of operators. Which operators are evaluated first and which last? There are many rules about this topic, which your compiler will define completely, but I would suggest that you don't worry about it at this point. Instead, use lots of parentheses to group variables, constants, and operators in a way meaningful to you. Parentheses always have the highest priority and will remove any question of which operations will be done first in any particular statements.

Goingon tothe next example ingroup three, 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 are true, leading to the result being "true", and "z" being assigned the value of 11. Note the since the variables, "r", "s", and "t" are "float" type variables, they could not be used this way, but they could each be compared to zero and the same type of expression could be used.

Continuing on to the fourth example of the third group 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".

4.13 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". The remaining parts of the compare are not evaluated, because it is an "AND" and 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, "z" will not be set to 4, and the variable "r" will not be changed.

4.14 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 "z" will not get set to the desired value, but for different reasons. In the case of the first one, 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. The second statement is much more straight forward because "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 last statement 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.

4.15 The Cryptic Part Of C

There are three constructs used in C that make no sense at all when first encountered because they are not intuitive, but they greatly increase the efficiency of the compiled code and are used extensively by experienced C programmers. You should therefore be exposed to them and learn touse them because they willappear inmost, if not all, of the programsyou seein the publications. Load and examine the file named `cryptic.c` for examples of the three new constructs.

```
main( )
{
int x = 0,y = 2,z = 1025;
float a = 0.0,b = 3.14159,c = -37.234;
   /* incrementing */
   x = x + 1;      /* This increments x  */
   x++;            /* This increments x  */
   ++x;            /* This increments x  */
   z = y++;        /* z = 2, y = 3  */
   z = ++y;        /* z = 4, y = 4  */
   /* decrementing */
   y = y - 1;  /* This decrements y  */
   y--;            /* This decrements y  */
   --y;            /* This decrements y  */
   y = 3;
   z = y--;        /* z = 3, y = 2  */
   z = --y;        /* z = 1, y = 1  */

   /* arithmetic op */
```

```
   a = a + 12;         /* This adds 12 to a  */
   a += 12;        /* This adds 12 more to a */
   a *= 3.2;       /* This multiplies a by 3.2  */
   a -= b;             /* This subtracts b from a  */
   a /= 10.0;      /* This divides a by 10.0  */

   /* conditional expression */
   a = (b >= 3.0 ? 2.0 ; 10.5 );     /*This expression      */
   if (b >= 3.0)                  /* And this expression */
       a = 2.0;                   /* are identical, both */
   else                           /* will cause the same */
        a = 10.5;                  /* result.             */
   c = (a > b?a:b);        /* c will have the max of a or b */
   c = (a > b?b:b);        /* c will have the min of a or b */
}
```

In this program, some variables are defined and initialized in the same statements for use below. The first 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 be 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 the next statement, 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, the value of "y" is incremented then its value is assigned to the variable "z".

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.

4.16 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 first statement of the "arithmetic operator" group of statements simply adds 12 to the value of the variable "a". The second statement does the same, but once again, it is not intuitive that they are the same. Any of the four basic functions of arithmetic, "+", "-", "x", or "/", can be handled in this way, by putting the function 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 liketheincrementing anddecrementing operators, thearithmetic operator is used extensively by experienced C programmers and it would pay you well to understand it.

4.17 The Conditional Expression

The conditional expression is just as cryptic as the last two, but once again it can be very useful so it would pay you to understand it. It consists of three expressions within parentheses separated by a question mark and a colon. The expression prior to the question mark is evaluated to determine if it is not true, the expression following the colon is evaluated. The result of the evaluation is used for the assignment. The final result is identical to that of an "if" statement with an "else" clause. This is illustrated by the second example in this group. The conditional expression has the added advantage of more compact code that will compile to fewer machine instructions in the final program.

The final two lines of this example program are given to illustrate a very compact way to assign the greater of two variables "a" or "b" to "c", and to assign the lessor of the same two variables to "c". Notice how efficient the code is in these two example.

4.18 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. I have found many functions that I wished to use within a program but needed a small modification to use it, requiring me 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 constantly exposed to them.

This has been a long chapter but it contained important material to get you started is 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.

4.19 Programming Exercises

1. Write a program that will count from 1 to 12 and print the count, and its square, for each count.

```
1       1
2       4
3       9
etc..
```

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.

```
1       1.00000
2        .50000
3        .33333
4        .25000
etc.
```

3. Write a program that will count from 1 to 100 and print only those values between 32 and 39, one to a line.

5
Functions and variables

5.1 Our First User Defined Function

Load and examine the file `sumsqres.c` for an example of a C program with functions.

```
int sum; /* This is a global variable */

main( )
{
int index;
   header();          /* This calls the function named header */
   for (index = 1;index <= 7;index++)
       square(index); /* This calls the square function     */
   ending();          /* This calls the ending function     */
}

header()              /* This is the function named header */
{
   sum = 0;           /* initialize the variable "sum" */
   printf("This is the header for the square program\n\n");
}

square(number)            /* This is the square function    */
int number;
{

int numsq;

   numsq = number * number;   /* This produces the square   */
   sum += numsq;
   printf("The square of %d is %d\n",number,numsq);
}

ending()   /* This is the ending function    */
{
   printf("\nThe sum of the squares is %d\n",sum);
}
```

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.

Notice the executable part of this program. It begins with a line 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, and finally a function named "ending "will be called and executed. For the moment ignore the "index" in the parentheses of the call to "square". We have seen that this program therefore calls a header, 7 square calls, and an ending. Now we need to define the functions.

5.2 Defining The Functions

Following the main program you will see another program 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 within the main program. Each of these statements are executed, and when they are all complete, control returns to the main program.

The first statement sets the variable "sum" equal to zero because we will use it to accumulate a sum of squares. Since the variable "sum" is defined as an integer type variable prior to the main program, it is available to be used in any of the following functions. It is called a "global" variable, and it's scope is the entire program and all functions. More will be said about the scope of variables at the end of this chapter. The next statement outputs a header message to the monitor. Program control then returns to the main programs in case there are no additional statements to execute in this function.

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 is a simple way. You will find functions to be very valuable in C programming.

5.3 Passing A Value To A Function

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++". You shouldget used to seeing this, as you will see it a lot in C programs.

In the call to the function "square", we have an added feature, namely the variable "index" within the parentheses. This is an indication to the compiler that when you go to the function, you wish to take along the value of index to use in the execution of the function. Looking ahead at the function "square", we find that another variable name is enclosed in its parentheses, namely the variable "number". This is the name we prefer to call the variable passed to the function when we are in the function. We can call it anything we wish as long as it follows the rules of naming an identifier. 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. Thus, the line containing "int number;" tells the function that the value passed to it will be an integer 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.

Following the opening brace of the function, we define another variable "numsq" for use only within the function itself, (more about that later) and proceed with the required calculations. We set "numsq" equal to the square of number, then add numsq to the current total stored in "sum". Remember that "sum += numsq" is the same as "sum = sum + numsq" from the last lesson. We print the number and its square, and return to the main program.

5.4 More About Passing A Value To A Function

When we passed the value of "index" to the function, a little more happened than meets the eye. We did not actually pass the value of index to the function, we actually passed a copy of the value. In this way the original value is protected from accidental corruption by a called function. We could have modified the variable "number" in any way we wished in the function "square", and when we returned to the main program, "index" would not have been modified. We thus protect the value of a variable in the main program from being accidentally corrupted, but we cannot return a value to the main program from a function using this technique. We will find a well defined method of returning values to the main program 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 program, we come to the last function call, the call to "ending". This call simply calls the last function which has no local variables defined. It prints out a message with the value of "sum" contained in it to end the program. The program ends by returning to the main program and finding nothing else to do. Compile and run this program and observe the output.

5.5 Now To Confess A Little Lie

I told you a short time ago that the only way to get a value back to the main program was through use of a global variable, but there is another way which we will discuss after you load and display the file named `squares.c`.

In this file 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.

```
main( )    /* This is the main program    */
{
int x,y;
   for(x = 0;x < 7;x++) {
       y = squ(x);     /* go get the value of x*x     */
       printf("The square of %d is %d\n",x,y);
   }
   for (x = 0;x <= 7;++x)
       printf("The value of %d is %d\n",x,squ(x));
}
squ(in)    /* function to get the value of in squared    */
int in;
{
int square;
   square = in * in;
   return(square); /* This sets squ() = square  */
}
```

In the main program, we define two integers and begin a "for" loop which will be executed 8 times. The first statement of 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 "squ" function taking along the value of "x" as a variable. Looking ahead to the function itself we find that the function prefers to call the variable "in" and it proceeds to square the value of "in" and call the result "square". Finally, a new kind of a statement appears, 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 "y" is then set equal to that value. If "x" were therefore assigned the value 4 prior to this call, "y" would then be set to 16 as a result of this line of code.

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 value of "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 which the function call is placed in the print 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 integer 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.

Compile and run this program.

5.6 Floating Point Functions

Load the program `floatsq.c` for an example of a function with a floating point type of return.

```
float z; /* This is a global variable */

main( )
{
int index;
float x,y,sqr(),glsqr();
   for (index = 0;index <= 7;index++){
       x = index;  /* convert int to float */
       y = sqr(x); /* square x to a floating point variable */
       printf("The square of %d is %10.4f\n",index,y);
   }
   for (index = 0; index <= 7;index++) {
       z = index;
       y = glsqr();
       printf("The square of %d is %10.4f\n",index,y);
   }
}

float sqr(inval)  /* square a float, return a float     */
float inval;
{
float square;
   square = inval * inval;
   return(square);
}

float glsqr() /* square a float, return a float */
{
   return(z*z);
}
```

It begins be defining a global floating point variable 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 and they are. This is the proper way in C to define that a function will return a value that is not of the 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".

Now refer to the function "sqr" near the center of the listing and you will see that the function name is preceded by the name "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 function 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".

The next function, namely "glsqr", will also return a "float" type variable, but it uses a global variable for input. It also does the squaring right within the return statement and therefore has no need to define a separate variable to store the product.

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.

There will be times that you will have a need for a function to return a pointer as a result of some calculation. There is a way to define a function so that it does just that. We haven't studied pointers yet, but we will soon. This is just a short preview of things to come.

5.7 Scope Of Variables

Load the next program, `scope.c` and display it for a discussion of the scope of variables in a program.

```
int count;            /* This is a global variable */

main( )
{
register int index;       /* This variable is available only in main   */
   head1();
   head2();
   head3();
                  /* main "for" loop of this program    */
   for (index = 8;index > 0;index--) {
       int stuff; /* This variable is only available in these braces */
       for (stuff = 0;stuff <= 6;stuff++)
           printf("%d",stuff);
       printf("  index is now %d\n",index);
   }
}
int counter;          /* This is available from this point on */
head1()
{
int index;            /* This variable is available only in head1 */
   index = 23;
   printf("The header1 value is %d\n",index);
}
head2()
{
int count;            /* This variable is available only in head2  */
                  /* and it displaces the global of the same name  */
   count = 53;
   printf("The header2 value is %d\n",count);
   counter = 77;
}
head3()
{
   printf("The header3 value is %d\n"counter);
}
```

The first variable defined is a global variable "count" which is available to any function in the program since it is defined before any of the functions. In addition, it is always available because it does not come and go as the program is executed. (That will make sense shortly.) Farther down in the program, another global variable named "counter" is defined which is also global but is not available to the main program since it is defined following the main program. 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.

Return to the main program and you will see the variable "index" defined as an integer. Ignore the word "register" for the moment. This variable is only available within the main program 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 program is always in operation, even when it gives control to another function. Another integer is defined

within the "for" braces, namely "stuff". Any pairing of braces can contain a variable definition which will be valid and available only while the program is executing statements within those braces. The variable will be an "automatic" variable and will cease to exist when execution leaves the braces. This is convenient to use for a loop counter or some other very localized variable.

5.8 More On "Automatic" Variables

Observe the function named "head1". It contains a variable named "index", which has nothing to do with the "index" of the main program, 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 variables is generated, and when "head1" completes its task, the variable "index" is eliminated completely from existence. Keep in mind however that this does not affect the variable of the same name in the main program, 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 to a function to the next call, the value of an automatic variable is not preserved and must therefore be reinitialized.

5.9 What Are Static Variables?

An additional variable type must be mentioned at this point, the "static" variable. By putting the reserved word "static" in front of a variable declaration within a function, the variable or variables in the declaration are static variables and will stay in existence from call to call of the particular function.

By putting the same reserved word 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 implies that it is possibletorefer to external variables inother separately compiledfiles, andthat is true. Examples of this usage will be given in chapter 14 of this tutorial.

5.10 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, 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 in this function. This allows you to write programs using existing functions without worrying about the variables that interface with the functions.

5.11 What Is A Register Variable?

Now to fulfil a promise made earlier about what a register variable is. A computer can keep data in a register or in a 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 if possible in order to speed up the execution of the program. Depending on the computer and the compiler, a small number of register variables may be allowed and are a desired feature. Check your compiler documentation for the availability of this feature and the number of register variables. Most compilers that do not have any register variables available, will simply ignore the word "register" and run normally, keeping all variables in memory.

Register variables are only available for use with integer and character type variables. This may or may not include some of the other integer-like variables such as unsigned, long, or short. Check the documentation for your compiler.

Register variables are allowed in HiTech C, with up to four non-pointer register variables (in 68000 registers D4 to D7), and up to three pointer register variables (in A3 to A5). This usage does not conflict with the 1616/OS usage of D0 to D2 and A0 to A2). As MS-DOS compilers typically only allow two register variables, many programmers do not make extensive use of register variables, so you can often gain a little extra speed when converting programs by a wideruse of register variables (the compiler will ignore the register variable request if no registers are available, and treat the variable as an ordinary one).

5.12 Where Do I Define Variables?

Now for a refinement on a general rule stated earlier. When you have variables brought to a function as arguments to the function, they are defined immediately after the function name and prior to the opening brace for the program. Other variables used in the function are defined at the beginning of the function, immediately following the opening brace of the function, and before any executable statements.

5.13 Standard Function Libraries

Every compiler comes with some standard predefined functions which are available for your use. These are mostly input/ouput functions, character and string manipulation functions, and math functions. We will cover most of these in subsequent chapters.

In addition, 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. The Applix equivalents are included in the header files mentioned elsewhere. These will notbe covered in any detail as you will be able to study the unique aspects of your compiler on your own. Many of these kinds of functions are used in the IBM versions of the example programs in chapter 14.

5.14 What Is Recursion?

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.

```
main( )
{
int index;
   index = 8;
   count_dn(index);
}
count_dn(count)
int count;
{
   count--;
   printf("The value of the count is %d\n",count);
   if (count > 0)
```

```
        count_dn(count);
    printf("Now the count is %d\n",count);
}
```

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 "index" is set to 8, and is used as the argument to the function "count_dn". The function simply decrements the variable, prints it out in a message, and if the variable is not zero, it calls itself, where it decrements it 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, until finally it will return to the main program and will return to DOS.

For purposes of understanding you can think of it as having 8 copies of the function "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.

5.15 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.

5.16 Another Example Of Recursion

The program named `backward.c` is another example of recursion, so load it and display it on your screen. This program is similar to the last one except that it uses a character array.

```
main( )
{
char line_of_char[80];
int index = 0;
   strcpy(line_of_char," this is a string.\n");
   /* the leading space in this */
   /* string is required so the */
   /* the last character "t" in */
   /* "this" is printed when    */
   /* the string is printed     */
   /* backwards due to the      */
   /* index being incremented   */
```

```
   /* to 1 before the the       */
   /* printf statement for      */
   /* printing the line back-   */
   /* wards                    */
   forward_and_backwards(line_of_char,index);
}
forward_and_backwards(line_of_char,index)
char line_of_char[];
int index;
{
   if (line_of_char[index]) {
       printf("%c",line_of_char[index]);
       index++;
       forward_and_backwards(line_of_char,index);
   }
   printf("%c",line_of_char[index]);
}
```

Each successive call to the function named "forward_and_backward" 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.

Don't worry about the character array defined in line 3 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.

One additional feature is built into this program in the IBM PC version. If you observe the two calls to the function, and the function itself, you will see that the function name is spelled three different ways in the last few characters in the original IBM version. The IBM compiler doesn't care how they are spelled because it only uses the first 8 characters of the function name so as far as it is concerned, the function is named "forward_". The remaining characters are simply ignored. If your compiler uses more than 8 characters as being significant, as does Hi-Tech, you will need to change two of the names so that all three names are identical, as we have done.

Compile and run this program and observe the results.

5.17 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 compiler will allow it.

6
Defines and Macros

6.1 Defines And Macros Are Aids To Clear Programming

Load and display the file named `define.c` for your first look at some defines and macros.

```
#define START  0  /* Starting point of loop */
#define ENDING 9  /* Ending point of loop  */
#define MAX(A,B)  ((A)>(B)?(A):(B))  /* Max macro definition */
#define MIN(A,B)  ((A)>(B)?(B):(A))  /* Min macro definition */

main( )
{
int index,mn,mx;
int count = 5;

   for (index = START;index <= ENDING;index++) {
       mx = MAX(index,count);
       mn = MIN(index,count);
       printf("Max is %d and min is %d\n",mx,mn);
   }
}
```

Notice the first four lines of the program each starting with the word "#define". This is the way all defines and macros are defined. 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 combination "START" is found and it will simply 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. It should be clear to you by now 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 the 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, but difficult, and possible disastrous if you missed one or two of the references.

In the same manner, the preprocessor will find all occurrence 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.

6.2 Is This Really Useful?

When we get to the chapters discussing input and output, we will need an indicator to tell us when we reach the end-of-file of an input file. Since different compilers use different numerical values for this, although most use either a zero or a minus 1, we will write the program with a "define" to define the EOF (end-of-file) used by our particular compiler. If at some later date, we change to a new compiler, it is a simple matter to change this one "define" to fix the entire program. End-of-line is another indicator that is not universal. This will make more sense when we get to the chapters on input and output.

6.3 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 the third line of the program on your screen 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 definition. Thus the first term will replace every "A" in the second definition and the second term will replace every "B" in the second definition. When line 12 of the program is reached, "index" will be substituted for every "A", and "count" will be substituted for every "B". 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". The results are then printed out. There are a lot of seemingly extra, they are essential. We will discuss the extra parentheses in our next program.

Compiler and run `define.c`.

6.4 Lets Look At A Wrong Macro

Load the file named `macro.c` and display it on your screen for a better look at a macro and its use.

```
#define WRONG(A) A*A*A           /* Wrong macro for cube    */
#define CUBE(A) (A)*(A)*(A)   /* Right macro for cube    */
#define SQUR(A) (A)*(A)       /* Right macro for square */
#define START 1
#define STOP 9

main( )
{
int i,offset;

   offset = 5;
   for (i = START;i <= STOP;i++) {
       printf("The square of %3d is %4d, and its cube is %6d\n",
               i+offset,SQUR(i+offset),CUBE(i+offset));
       printf("The wrong of  %3d is %6d\n",i+offset,WRONG(i+offset));
   }
}
```

The first line defines a macro named "WRONG" that appears to get 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 all cases.

Consider the program itself where the CUBE of i+offset is calculated. 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.

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

6.5 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.

7
Strings and Arrays

7.1 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 titles and names, and is esthetically 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 definitionis a series of "char" type data terminated by a NULL character, which is a zero.

When C is going to use a string of data in some way, either to compare it with another, 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.

7.2 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.

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.

```
main( )
{
char name[5];       /* define a string of characters */
   name[0] = 'D';
   name[1] = 'a';
   name[2] = 'v';
   name[3] = 'e';
   name[4] = 0;      /* Null character - end of text  */
   printf("The name is %s\n",name);
   printf("One letter is %c\n",name[2]);
   printf("Part of the name is %s\n",&name[1]);
}
```

The first thing new is the line that defined a "char" type of data entity. The square brackets define an array subscript in C, and in the case of the data definition statement, the 5 in the brackets defines 5 data fields of type "char" all defined as the variable "name". In the C language, all subscripts start at 0 and increase by 1 each step up to the maximum which in this case is 4. We therefore 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.

7.3 How Do We Use The String?

The variable "name" is therefore a string which can hold up to 5 characters, since we need room for the NULL character, there are actually only four useful characters. To load something useful into the string, we have 5 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 "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, we will simply print it out with some other string data

in the output statement. The % is the output definition to output a string and the system will output characters starting with the first one in "name" until it comes to the NULL character, and it will quit. Notice that in the "printf" statement, only the variable name "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.)

7.4 Outputting Part Of A String

The next "printf" illustrates that we can output any single character of the string by using the "%c" and naming the particular character of "name" we want by including the subscript. 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.

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. That is an incorrect conclusion because strings are very easy to use as we will see in the next example program.

Compile and run this program.

7.5 Some String Subroutines

Load the example program `strings.c` for an example of some ways to use strings.

```
main( )
{
char name1[12],name2[12],mixed[25];
char title[20];

   strcpy(name1,"Rosalinda");
   strcpy(name2,"Zeke");
   strcpy(title,"This is the title.");

   printf("       %s\n\n"title);
   printf("Name 1 is %s\n",name1);
   printf(Name 2 is %s\n",name2);

   if(strcmp(name1,name2)>0)  /* return 1 if name1 > name2 */
       strcpy(mixed,name1);
   else
       strcpy(mixed,name2);

   printf("The biggest name alphabetically is %s\n",mixed);

   strcpy(mixed,name1);
   strcat(mixed,"  ");
   strcat(mixed,name2);
   printf("Both names are %s\n",mixed);
}
```

First we define four strings. 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. It is easy to remember which one gets copies to which is 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, name1 will contain the string "Rosalinda", but without the double quotes, they are the complier's way of knowing that you are defining a string.

Likewise, "Zeke" is copied into "name2" by the second statement, then the "title" is copied. The title and both names are then printed out. Note that it is not necessary for the defined string tobe exactly the same size as the string it will be called upon to store, only that it is at least as longas the string plus one more character for the NULL.

7.6 Alphabetical Sorting Of Strings

The next function we will look at is the "strcmp" or the string compare function. It will return a 1 it 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 variable "mixed", and the largest name alphabetically is printed out. It should come as no surprise to you that "Zeke" wins because it is alphabetically larger, length doesn't matter, only 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 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.

7.7 Combining Strings

The last four statements have 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 variable "mixed".

Strings are not difficult and are extremely useful. You should spend some time getting familiar with them before proceeding on to the next topic.

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

7.8 An Array Of Integers

Load the file `intarray.c` and display it on your monitor for an example of an array of integers.

```
main( )
{
int values[12];
int index;
   for (index = 0;index < 12;index++)
       values[index] = 2 * (index + 4);
   for (index = 0;index < 12;index++)
            printf("The value at index = %2d is %3d\n",index,values[in-
dex]);
}
```

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 section. We have 12 integer variables to work, with not counting the one named "index". The names of the variables are "values[0]", "values[1]", ... , and "values[11]". Next we have a loop to assign nonsense, but well defined, data to each of the 12 variables, then print all 12 out. You should have no trouble following this program, but be sure you understand it. Compile and run it to see if it does what you expect it to do.

7.9 An Array Of Floating Point Data

Load and display the program named `bigarray.c` for an example of a program with an array of "float" type data.

```
/* NOTE
 This example has been modified to use another integer type variable
instead
 of a float type variable due to the problems in the float library with the
 HI-TECH C Compiler. The compiler returns no fatal errors when the type
float
 variable is used and the float library included at link time but the Xrel
file
 produces a address error. The original code is commented out so that you
can
 still see that was intended in the tutorial.
*/
char name1[ ] = "First Program Title";
main()
{
int index;
int stuff[12];
/* float weird[12]; */
int weird[12];
static char name2[ ] = "Second Program Title";
    for  (index = 0;index < 12;index++) {
        stuff[index] = index + 10;
  /*   weird[index] = 12.0  *  (index  +  7); */
        weird[index] = 12 * (index  +  7);
    }
    printf("%s\n",name1);
    printf("%s\n\n",name2);
    for (index = 0;index< 12;index++) {
       printf("%5d %5d   %5d\n",index,stuff[index],weird[index]);
   /*  printf("%5d %5d %10.3f\n",index,stuff[index],weird[index]); */
    }
}
```

This program has an extra feature to illustrate how strings can be initialized. The first line of the program illustrates to you 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. Another string is initialized in 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.

7.10 Getting Data Back From A Function

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. Load the program `passback.c` for an example of doing that.

```
main( )
{
int index;
int matrix[20];
   for (index = 0;index < 20;index++)           /* generate data  */
       matrix[index] = index + 1;
   for (index = 0;index < 5;index++)         /* print original data */
       printf("Start  matrix[%d] = %d\n",index,matrix[index]);
```

```
   dosome(matrix);            /* go to a function & modify matrix */
   for (index = 0;index < 5;index++)         /* print modified matrix  */
       printf("back matrix[%d] = %d\n",index,matrix[index]);
}
dosome(list)      /* This will illustrate returning data    */
int list[];
{
int i;
   for (i = 0;i < 5;i++)
       printf("Before marrix[%d] = %d\n",i,list[i]);
   for (i = 0;i < 20;i++)             /* add 10 to all values    */
       list[i] +=10;
   for (i =0;i < 5;i++)           /* print modified matrix */
       printf("After  matrix[%d] = %d\n",i,list[i]);
}
```

In this program, we define an array of 20 variables named "matrix", then assign some nonsense data to the variables, and print out the first five. Then we call the function "dosome" taking along the entire array by putting the name of the array in the parentheses.

The function "dosome" has a name in its parentheses also but it prefers to call the array "list". The function needs to be told that it is really getting an array passed to it and that the array is of type "int". The following line, prior to the bracket which starts the program, 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, but you could if you so desired. 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. On 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. We print out the first 5 again to see if they did indeed get passed here. Then we add ten to each of the elements and print out the new values. Finally we return to the main programand print out the same 5 data points. We find that we have indeed modified the data in the function, and when we returned to the main program, we brought the changes back. Compileand run this program to verify this conclusion.

7.11 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 array is passed 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.

7.12 A Hint At A Future Lesson

Another way of getting data back from a function to the calling program is by using pointers which we will cover in the next chapter. When we get there we will find that 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 pretty elaborate arrays.

7.13 Multiply Dimensioned Arrays

Load and display the file named `multiary.c` for an example of a program with doubly dimensioned arrays.

```
main( )
{
int i,j;
int big[8][8],huge[25][12];
   for (i = 0;i < 8;i++)
       for (j = 0;j < 8;j++)
           big[i][j] = i * j;     /* This is a multiplication table     */
   for (i = 0;i < 25;i++)
       for (j = 0;j < 12;j++)
           huge[i][j] = i + j;    /* This is an addition table  */
   big[2][6] = huge[24][10] *22;
   big[2][2] = 5;
   big[big][2][2]...big[2][2]. = 177;    /* this is big[5][5] = 177;    */
   for (i = 0;i < 8;i++) {
       for (j = 0;j < 8;j++)
           printf("%5d ",big[i][j]);
       printf("\n");          /* newline for each increase in i     */
   }
}
```

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 "huge" is also defined which is not square to illustrate that the array need not be square. Both are filled up 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 "huge" after being multiplied by 22. Next "big[2][2]" is assigned the arbitrary value of 5, and this value is used for the subscripts of the next assignment statement. The third assignment statement 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 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 so you can check the values to see if they did get set the way you expected them to.

7.14 Programming Exercises

1. Write a program with three short strings, about 6 characters each, and use "strcpy" to copy "one", "two", and "three" into them. Concatenate the three strings into one string 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.
1 2 + 10 = 12
2 4 + 20 = 24
3 6 + 30 = 36 etc.

Hint; The print statement will be similar to:
printf("%4d %4d + %4d = %4d\n",index,array1[index], array2[index],arrays[index]);

8
Pointers

8.1 What Is A Pointer?

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.

```
main( )              /*  illustratrion of pointer use  */
{
int index,*pt1,*pt2;
   index = 39;           /*  any numerical value    */
   pt1 = &index;         /* the address of index  */
   pt2 = pt1;
   printf("The value is %d %d %d\n",index,*pt1,*pt2);
   *pt1 = 13;     /* this changes the value of index    */
   printf("The value is %d %d %d\n",index,*pt1,*pt2);
}
```

For the moment, ignore the declaration 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 first statement, it should be clear that we assign the value of 39 to the variable "index". This is no surprise, we have been doing it for several programs now. The next statement 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 "index" is a simple variable. Now we have 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.

8.2 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 six 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 nine of the program can be read as "The stored (starred) value to which the pointer "pt1" points is assigned the value 13". Now you can see why it is convenient to think of the asterisk as a star, it sort of sounds like the word store.

8.3 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 them 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. Line six of the program assigns the pointer "pt1" to point to the variable we have already defined as "index" to "pt1". Since we have a pointer to "index", we can manipulate the value of "index" by using either the variable name itself, or the pointer.

Line nine modifies the value by using the pointer. Since the pointer "pt1" points to the variable "index", then putting a star in front of the pointer name refers to the memory location to which it is pointing. Line nine therefore assigns to "index" the value of 13. 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.

8.4 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 seven, it doesn't point to anything, it contains garbage. Of course, that is also true of any variable until a value is assigned to it. Statement seven assigns "pt2" the same address as "pt1", so that now "pt2" also points to the variable "index". So 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 identical in meaning. This fact is illustrated in the first "printf" statement since this statement uses the three means of identifying the same variable to print out the same variable three times.

8.5 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 point to the single variable. This is illustrated in the next statement which assigns the value of 13 to the variable "index", because that is where the pointer "pt1" is pointing. The next "printf" statement 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, not three.

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.

8.6 How Do You Declare A Pointer?

Now to keep a promise and tell you how to declare a pointer. Refer to the third line 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.

A pointer must be defined to point to some 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. In the same manner that a "float" type of variable cannot be added to an "int" type variable, a pointer to a "float" variable cannot be used to point to an integer variable.

Compile and run this program and observe that there is only one variable and the single statement in line 9 changes the one variable which is displayed three times.

8.7 The Second Program With Pointers

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.

```
main( )
{
char strg[40],*there,one,two;
int *pt,list[100],index;

   strcpy(strg,"This is a character string.");

   one = strg[0];         /* one and two are identical */
   two = *strg;
   printf("The first output is %c %c\n",one,two);

   one = strg[8];         /* one and two are identical */
   two = *(strg+8);
   printf("The second output is %c %c %c\n",one,two);

   there = strg+10;       /* strg+10 is identical to strg[10]  */
   printf("The third output is %c\n",strg[10]);
   printf("The fourth output is %c\n",*there);

   for (index = 0;index < 100;index++)
       list[index] = index + 100;

   pt = list + 27;
   printf("The fifth output is %d\n",list[27]);
   printf("The sixth output is %d\n",*pt);
}
```

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 variable named "strg" and "list". We will use them to show the correspondence between pointers and array names.

8.8 A String Variable Is Actually A Pointer

In theprogramming language C, a string variable is definedto be simplya pointer to the beginning of a string. This will take some explaining. Refer to the example program on your monitor. You will notice that first 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 pointer by definition of the C language, we can assign the same value to "two" by using the star and the string name. The result if 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 the ninth line as "two = *strg[0];" because the star takes the place of the square brackets.

For all practical purposes, "strg" is a pointer. 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 initial value and therefore always points to its string. It could be thought of as a pointer constant, and in some applications you may desire a pointer that cannot be corrupted in any way. 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 12, the variable "one" is assigned the value of the ninth variable (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".

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 variable 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 stored. When we get to the chapter on structures, we will see that a variable can have many, even into the hundreds or thousands, of characters per variable, but the indexing will be handled automatically for us by the system.

Since "there" is already a pointer, it can be assigned the value of the eleventh element of "strg" by the statement in line 16 of the program. Remember that since "there" is a true pointer, 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".

8.9 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 adda constant to anaddress, therebymoving 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 this 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.

8.10 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. Next we assign the pointer "pt" the value 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.

Compile and run `pointer2.c` and study the output.

8.11 Function Data Return With A Pointer

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 program named `twoway.c` for an example of this.

```
main( )
{
int pecans,apples;
   pecans = 100;
   apples = 101;
   printf("The starting values are %d %d\n",pecans,apples);
                       /* when we call "fixup"          */
   fixup(pecans,&apples);    /* we take the value of pecans   */
                       /* we take the address of apples */
   printf("The ending values are %d %d\n",pecans,apples);
}
```

```
fixup(nuts,fruit)             /* nuts is an integer value    */
int nuts,*fruit;          /* fruit points to an integer */
{
   printf("The value are %d %d\n",nuts,*fruit);
   nuts = 135;
   *fruit = 172;
   printf("The values are %d %d\n",nuts,*fruit);
}
```

In `twoway.c`, there are two variables defined in the main program "pecans" and "apples". Notice that neither of these if defined as a pointer. We assign values to both of these and print them out, then call the function "fixup" taking with us both of these values. 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 integervariable and a pointer to an integer variable. This turns out to be very simple. Notice that theparameter definitions in the function define "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 headingand 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 program and print out the two values again. We will find that the value of pecans will be restored to its value before 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. 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 copy, we had access to the original variable and could change it in the function. When we returned to the main program, we found a changed value in "apples" when we printed it out.

By using a pointer in a function call, we can have access to the data in the function and change it in such a way that when we return to the calling program, we have a changed value of data. It must be pointed out however, that if you modify the value of the pointer itself in the function, you will have restored pointer when you return because the pointer you use in the function is a copy of the original. 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.

8.12 Pointers Are Valuable

Even though you are probably somewhat intimidated at this point by the 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.

8.13 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 to print out the string backwards by pointing to the end and using a decrementing pointer.

9
Standard Input/Output

9.1 The Stdio.H Header File

Load the file `simpleio.c` for our first look at a file with standard I/O. Standard I/O refersto the most usual places where 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 startto use them so lets look at the file in front of you.

```
# include "/sys/stdio.h"     /* standard header for input/output */

main( )
{
char c;
   printf("Enter any characters, X = halt program.\n");
   do {
       c = getchar();     /* Get a character from the kb  */
       putchar(c);        /* Display the character on the monitor  */
     }  while (c != 'X');     /* Until and X is hit  */
   printf("\nEnd of program.\n");
}
```

The first thing you notice is the first line of the file, the #include "stdio.h" line. 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. This particular file is composed of several standard #defines 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 compiler. Each of the header files has a specific purpose and any or all of them can be included in any program.

Most C compilers use the double quote marks to indicate that the "include" file will be found in the current directory. A few use the "less than" and "greater than" signs to indicate that the file will be found in a standard header file. Nearly all MSDOS C compilers use the double quotes, and most require the "include" file to be in the default directory. All of the programs in this tutorial have the double quotes in the "include" statements. If your compiler uses the other notation, you will have to change them before compiling.

9.2 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 nearly 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. You will often, when reading literature about C, find a reference to K & R. This refers to the book written by Kernigan and Ritchie. You would be advised to purchase a copy for reference.

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 does. It defines the standard input and output functions in the form of #defines and macros. 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.

9.3 Other Include Files

When you begin writing larger programs and splitting them up into separately compiled portions, you will have occasion to use some statements common to each of the portions. It would be to your advantage to make a separate file containing the statements and use the #include to insert it into each of the files. If you want to change any file, 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.

9.4 Back To The File Named Simpleio.c

Let us continue our tour of the file in question. The one variable "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 "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 the new functions. These are functions to read a character from the keyboard and display it on the monitor one character at a time.

The function "getchar()" reads a single character from the standard input device, the keyboard being assumed because that is the standard input device, and assigns it to the variable "c". The next function "putchar(c)", uses the standard output device, the video monitor, and outputs the character contained in the variable "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.

At this point we need to mention that 1616/OS, Unix and MS-DOS sometimes do different things, even in apparently simple areas such as this. In some version of HiTech C for the 1616/OS, characters are repeated as you type them, but are not repeated when you press the return key. This is because the "getchar()" function was redefined as the 1616/OS function "getch()", which is not precisely the same.

Compile and run this program on an MS-DOS machine 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. In fact, 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.

9.5 Dos Is Helping Us Out (Or Getting In The Way)

We need to understand a little bit about how MS-DOS works to understand what is happening here. When data is read from the keyboard, under MS-DOS control, 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. While 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 MS-DOS, and when you return the carriage, 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 under MS-DOS, echoed to the monitor, and placed in the MS-DOS keyboard input buffer. MS-DOS 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.

In the Applix 1616, the `stdio.h` function getchar() is defined in `stdio.h` as getch(), which is provided by 1616/OS, but is not exactly identical to the expected reslts from "getchar()". The keyboard buffer is read by getch() and the characters passed straight along to the program. Within the program, the characters are echoed to the display by putchar(), and the 1616/OS only has to change the cursor position. If you leave out the #include "stdi.h", then the HiTech C compiler will use its default "getchar()" function, which is correct, and will work the same as MS-DOS

Compileand run "`simpleio.c`. After running the programseveral times andfeeling confidant that you understand the above explanation, we will go on to another program.

Don't get discouraged by the above seemingly weird behaviour 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 probably find some of the following useful also.

9.6 Another Strange I/O Method

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, 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.

```
# include "/sys/stdio.h"     /* standard header for input/output */
main( )
{
char c;
   printf("Enter any characters, terminate program with X\n");
   do {
       c = getch();       /* Get a character  */
       putchar(c);        /* Display the hit key    */
     }  while (c != 'X');
     printf("\nEnd of program.\n");
}
```

The "getch()" is a new function that is a "get character" function. It differs from "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 then reads a character immediately displays it on the screen, and continues the operation until a capital X is typed. You wil recognise that using this "getch()" function accidently in the #include "stdio.h" caused problems in the previous 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. We do have another problem here, there is no linefeed with the carriage return.

9.7 Now We Need A Line Feed

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.

```
# include "stdio.h"
# define CR 13        /* this defines CR to be 13 */
# define LF 10        /* this defines LF to be 10 */

main( )
{
char c;
    printf("input any characters,hit X to stop.\n");

    do {
       c = getch();                  /* get a character */
       putchar(c);                   /* display the hit key */
       if (c == CR) putchar(LF);     /* if it is a carriage return
                                        put out a linefeed too */
    } while (c != 'X');

    printf("\nEnd of program.\n");
}
```

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, we also output a linefeed which is the LF. We could have just as well have left out the two #define statements and used "if (c == 13) putchar(10);" but it would not be very descriptive of what we are doing here. The method used in the program represents better programming practice.

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 you are using a nonstandard compiler, it may not find a "CR" because your system returns a "LF" character to indicate end-of-line. It will be up to you to determine what method your compiler uses. The quickest way is to add a "printf" statement that prints the input character in decimal format.

9.8 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. Lets take a look at each.

When using the first method, 1616/OS is actually doing all of the work for us, by storing the characters in an input buffer and signalling us when a full line has been entered. We could writea program that, for example, did a lot of calculations, then went to get some input. While we were doing the calculations, 1616/OS 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 1616/OS 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 1616/OS 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. 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.

The discussion so fact 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.

9.9 Now To Read In Some Integers

Load and display the file named `intin.c` for an example of reading in some formatted data. 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 some how acquires the value of 100.

```
#include "/sys/stdio.h"
main( )
{
int valin;
   printf("Input a number from 0 to 2 billion, stop with 100.\n");
   do {
       scanf("%d",&valin);    /* read a single integer value in */
       printf("The value is %d\n",valin);
   } while (valin  != 100);
   printf("End of program\n");
}
```

Instead of reading in a character at a time, as we have in the last three files, 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 for the function. Recall that a function must have the address of a variable in order to return the value to the calling program. Failing to supply a pointer in the "scanf" function is probably 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 1616/OS 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 reading 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 to 100 will cause the program to terminate, and the increased responsibility you must assume using C rather than a higher level language such as Pascal, Modula-2, etc.

The above paragraph is true for the HiTech C compiler for the Applix, which uses 32 bit integers. MS-DOS compilers typically use an integer value of 16 bits, and thus accept input values only up to 32767. If your compiler is different, the same principles will be true but at 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 non-decimal characters.

9.10 Character String Input

Load and display the file named `stringin.c` for an example of reading a string variable. 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).

```
#include "/sys/stdio.h"
main( )
{
char big[25];
   printf("Input a character string,up to 25 characters.\n");
   printf("An X in column 1 causes the program to stop.\n");
   do      {
       scanf("%s",big);
       printf("The string is -> %s\n",big);
   }   while (big[0] !='X');
   printf("End of program.\n");
}
```

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 except.

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 DOS 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 program stops.

Try entering more than 24 characters to see what the program does. It should generate an error, but that will be highly dependent on the system you are using. 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. It is, but you also get a lot of flexibility in the bargain too.

9.11 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 to 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".

9.12 In Memory I/O

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.

```
main( )
{
int numbers[5], result[5], index;
char line[80];

   number[0] = 74;
   number[1] = 18;
   number[2] = 33;
   number[3] = 30;
   number[4] = 97;
   sprintf(line,%d %d      %d %d %d\n",numbers[0],numbers[1],
           numbers[2],numbers[3],numbers[4]);
   printf("%s",line);
   sscanf(line,"%d %d %d %d       %d",&result[4],&result[3],
           (result+2),(result+1),result);
   for (index = 0;index < 5;index++)

   printf("The final result is %d\n",result[index]);
}
```

In `inmem.c`, we define a few variables, then assign some values to the ones named "numbers" for illustrative purposes and then use a "sprintf" 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 "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 an+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 outside the computer. 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 many ways to declare a pointer several methods are used, but all are pointers. The first two simply declare

the address of the elements of the arrays, 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.

9.13 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 in 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.

9.14 Standard Error Output

Sometimesit is desirable to redirect the outputfrom the standard output device toa 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 in MS-DOS systems. Load and display `special.c` for an example of this new function.

```
#include "/sys/stdio.h"

main( )
{
int index;
   for (index = 0;index < 6;index++) {
       printf("This line goes to standard output.\n");
       fprintf(stderr,"This line goes to the error device.\n");
   }
   exit(4);
       /* This can be tested with the MS-DOS errorlevel
          command in a batch file. The number returned
          is used as follows;

          IF ERRORLEVEL 4 GOTO FOUR
          (continue here if less than 4)
          .
          .
          GOTO DONE
          :FOUR
          (continue here if 4 or greater)
          .
          .
          :DONE
               */
}
```

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 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 a 1616/OS prompt;

F0/ special >stuff

More information about I/O redirection can be found in your 1616/OS *Users Manual*. 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 the file named "STUFF" containing 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.

9.15 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 simply exits the program and returns the value of 4 to MS-DOS. Any number from 0 to 9 can be used in the parentheses for DOS communication. Ifyou are operating in a BATCH file, this number can be tested with the "ERRORLEVEL" command. Note that these characteristics only apply to MS-DOS.

Most compilers that operate in several passes return a 1 with this mechanism to indicate that a fatal error has occurred and it would be a waste of time to go on to another pass resulting in even more errors.

It is therefore wise to use a batch file for compiling programs and testing the returned value for errors.

9.16 Programming Exercise

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 in MS-DOS systems, the first being a zero which is the indication to the system that a special key was hit. This won't happen with the 1616/OS. Function keys do return leading zeros (since this is done by the microprocessor actually inside the keyboard), however the 1616/OS keyboard driver does not pass them along to the program. The way to getaround this is touse syscall .29 to redefinefunction keys, so that they pass whatever character string you desire to the program. See the file `vc.c` function initfk() for a clean example of this.

10
File Input/Output

10.1 Output To A File

Load and display the file named `formout.c` for your first example of writing data to a file.

```
#include "/sys/stdio.h"
main( )
{
FILE *fp;
char stuff[25];
int index;

   fp = fopen("TENLINES.TXT","w");   /* open for writing    */
   strcpy(stuff,"This is an example line.");

   for (index = 1;index <= 10;index++)
       fprintf(fp,"%s Line number %d\n",stuff,index);

   fclose(fp);    /* close the file before ending program  */
}
```

We begin as before with the "include" statement for "stdio.h", then define some variables for use in the example including a rather strange looking new type.

The type "FILE" is used for a file variable 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 contains the requirement for a pointer to a "FILE", and as usual, the name can be any valid variable name.

10.2 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 the first line of the program. The file pointer, "fp" in our case, points to the file and two arguments are required in the parentheses, the filename first, followed by the file type. The filename is any valid 1616/OS filename, 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 erased. If you don’t have a file by this name, that is good because we will create one and put some data into it.

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. 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.

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.

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 the end of the present data so that any new data will be added to any data that already exists in the file.

10.3 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.

10.4 Closing A File

To close a file, you simply 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 DOS. 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 at your directory for a file named TENLINES.TXT and "type" it. That is where your output will be. Compare the output with that specified in the program. It should agree.

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

10.5 Outputting A Single Character At A Time

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.

```
#include  "/sys/stdio.h"
main()
{
FILE *point;
char others[35];
int indexer,count;
   strcpy(others,"Additional lines.");
   point = fopen("tenlines.txt","a");  /* open for appending */
   for (count = 1;count <= 10;count++) {
       for (indexer = 0;others[indexer];indexer++)
           putc(others[indexer],point);  /* output a single character  */
       putc('\n',point);                 /* output a linefeed  */
   }
   fclose(point);
}
```

The program begins with the "include" statement, then defines some variables including a file pointer. We have called the file pointer "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 in the "fopen" function, except this time we use the lower cases for the filename. This is done simply to illustrate that DOS doesn't care about the case of the filename. Notice that the file will be opened for appending so we will add to the lines inserted during the last program.

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 "others" is detected to be a zero.

10.6 The "Putc" Function

The part of the program we are interested in is the "putc" function. It outputs one character ata 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.

Following execution of the program, "type" 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.

10.7 Reading A File

Load the file named `readchar.c` and display it on your monitor. This is our first program to read a file.

```
#include "/sys/stdio.h"

main( )
{
FILE *funny;
int c;

   funny = fopen("TENLINES.TXT","r");

   if (funny == NULL) printf("File doesn't exist\n");
   else {
       do {
           c = getc(funny);   /* get one character from the file    */
           putchar(c);        /* display it on the monitor      */
       } while (c != EOF);        /* repeat until EOF (end of file)     */
   }
   fclose(funny);
}
```

This program begins with the familiar "include", 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 it. 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, we print a message and quit. If the file does not exist, the system will set the pointer equal to NULL which we can test.

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 could use a "char" variable for this purpose. There is a problem with that however, because on some, if not most, implementations of C, the EOF returns a minus one which a "char" type variable is not capable of containing. A "char" type variable can only have the values of zero to 255, so it will return a 255 for a minus one on those compilers that use a minus one for EOF. This is a very frustrating problem to try to find because no diagnostic is given. The program simply can never find the EOF and will therefore never terminate the loop. This is easy to prevent, always use an "int" type variable for use in returning an EOF. You can tell what your compiler uses for EOF by looking at the "stdio.h" file where EOF is defined. That is the standard place to define such values.

There is 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".

10.8 Reading A Word At A Time

Load and display the file named `readtext.c` for an example of how to read a word at a time.

```
#include "/sys/stdio.h"

main( )
{
FILE *fp1;
char oneword[100];
int c;
   fp1 = fopen("TENLINES.TXT","r");

   do {
       c = fscanf(fp1,"%s",oneword); /* got one word from the file */
       printf("%s\n",oneword);       /* display it on the monitor  */
   } while (c != EOF);               /* repeat until EOF           */
   fclose(fp1);
}
```

This program is nearly identical as 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.

10.9 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 "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 original 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 because of the "printf" before checking for EOF.

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.

10.10 Finally, We Read A Full Line

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 for the addition of a new quantity, the NULL.

```
#include "/sys/stdio.h"

main( )
{
FILE *fp1;
char oneword[100];
char *c;

   fp1 = fopen("TENLINES.TXT","r");

   do {
       c = fgets(oneword,100,fp1);   /* get one line from the file */
       if (c != NULL);
           printf("%s",oneword);  /* display it on the monitor  */
   } while (c != NULL);           /* repeat until NULL      */

   fclose(fp1);
}
```

We are using "fgets" which reads in 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 "c" will be assigned the value of NULL. NULL is defined as zero in your "stdio.h" file.

When we find that "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.

10.11 How To Use A Variable Filename

Load and display the file `anyfile.c` for an example of reading from any file. This program asks the user for the filename desired, reads in the filename and 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.

```
#include  "stdio.h"
```

```
main( )
{
FILE  *fp1;
char oneword[100],filename[25];
char *c;

    printf("enter filename ->  ");
    scanf("%s",filename);          /* read the desired filename */
    fp1 = fopen(filename,"r");

    do {
       c = fgets(oneword,100,fp1); /* get one line from the file */
       if (c != NULL)
          printf("%s",oneword);    /* display it on the monitor  */
       } while (c != NULL);        /* repeat until NULL          */

       fclose(fp1);
}
```

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.

10.12 How Do We Print?

Load the last example file 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.

```
#include "/sys/stdio.h"

main( )
{
FILE *funny,*printer;
int c;

   funny = fopen("TENLINES.TXT","r");    /* open input file     */
   printer = fopen("PRN","w");       /* open printer file */

   do {
       c = getc(funny);   /* got one character from the file */
       if (c != EOF) {
           putchar(c);    /* display it on the monitor */
           putc(c,printer); /* print the character   */
       }
       } while (c != EOF);       /* repeat until EOF (end of file)  */

   fclose(funny);
   fclose(printer);
}
```

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. There are no definite standards as far as the name or names to be used for the printer, but the 1616/OS names are, "CENT:", "SA:", and "SB:". Check your documentation for your particular implementation.

Some of the newest MS-DOS 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.

You can now erase TENLINES.TXT from your disk. We will not be using it in any of the later chapters.

10.13 Programming Exercises

1. Write a program that will prompt for a filename for a read 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.

11
Structures and Unions

11.1 What Is A Structure?

A structure is a user defined data type. 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. An easy to understand definition is, a structure is a grouping of related data in a way convenientto the programmer or user of the program. The best way to understand a structure is to look atan example, so if you will load and display `struct1.c`, we will do just that.

```
main( )
{
struct {
   char initial; /* last name initial  */int
   age;       /* childs age              */
   int grade; /* childs grade in school */
   } boy,girl;
   boy.initial = 'R';
   boy.age = 15;
   boy.grade = 75;
   girl.age = boy.age - 1;    /* she is one year younger  */
   girl.grade = 82;
   girl.initial = 'H';
   printf("%c is %d years old and got a grade of %d\n",
          girl.initial,girl.age,girl.grade);
   printf("%c is %d years old and got a grade of %d\n",
          boy.initial, boy.age, boy.grade);
}
```

The program begins with a structure definition. The key word "struct" is followed by some simple variables between the braces, which are the components of the structure. After the closing brace, you will find two variables listed, namely "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 "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.

11.2 A Single Compound Variable

Let us examine the variable "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 integer variable and can therefore be used anywhere in a C program where it is legal to use an integer variable, in calculations, as a counter, in I/O operations, etc. The only problem we have is defining how to use the simple variable "age" which is a part of the compound variable "boy". 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.

11.3 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 assigned that in the structure, so it must be assigned a character of data. Notice that "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 assigned values but in a different order to illustrate that the order of assignment is not critical.

11.4 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.

11.5 An Array Of Structures

Load and display the next program named `struct2.c`.

```
main( )
{
struct {
   char initial;
   int age;
   int grade;
   } kids[12];
int index;
   for (index = 0;index < 12;index++) {
       kids[index].initial = 'A' + index;
       kids[index].age = 16;
       kids[index].grade = 84;
   }
   kids[3].age = kids[5].age = 17;
   kids[2].grade = kids[6].grade = 92;
   kids[4].grade = 57;
   for (index = 0;index < 12;index++)
       printf("%c is %d years old and got a grade of %d\n",
           kids[index].initial,kids[index].age,
           kids[index].grade);
}
```

This program contains the same structure definition as before but this time we define an array of 12 variables named "kids". This program therefore 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 three of the fields. 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. You might consider this the crude beginning of a data base, which it is.

In the next few instructions of the 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.

11.6 A Note To Pascal Programmers

Pascal allows you to copy an entire RECORD with one statement. This is not possible in C. You must copy each element of a structure one at a time. As improvements to the language are defined, this will be one of the refinements. In fact, some of the newer compilers already allow structure assignment. Check your compiler documentation to see if your compiler has this feature yet.

11.7 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.

11.8 Using Pointers And Structures Together

Load and display the file named `struct3.c` for an example of using pointers with structures. Thisprogram isidentical tothe last program except that it uses pointersfor some of the operations.

```
main( )
{
struct  {
   char initial;
   int age;
   int grade;
   } kids[12],*point;
int index;
   for (index = 0;index < 12;index++) {
       point = kids + index;
       point->initial = 'A' + index;
       point->age = 16;
       point->grade = 84;
   }
   kids[3].age = kids[5].age = 17;
   kids[2].grade = kids[6].grade = 92;
   kids[4].grade = 57;
   for (index = 0;index < 12;index++) {
       point = kids + index;
       printf("%c is %d years old and got a grade of %d\n",
           (*point).initial,kids[index].age,
           point->grade);
   }
}
```

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.

The next difference is in the for loop where we use the pointer for accessing the data fields. Since "kids" is a pointer variable that points to the structure, we can define "point" in terms of "kids". The variable "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 it will point to the first element of the array, a structure containing three fields.

11.9 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 advancethe 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 the beginning of one of the array elements each time. We can therefore use the pointer to reference the various elements of the structure. Referring to the elements of a structure with a pointer occurs so often in C that a special method of doing that 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 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.

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, and in the for loop used for output at the end of the program, we use three different methods. 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.

Compile and run this program.

11.10 Nested And Named Structures

Load and display 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 of definition. This will be illustrated with the program on your monitor.

```
main( )
{
struct person {
   char name[25];
   int age;
   char status;       /* M = married, S = single */
} ;
```

```
   struct alldat {
   int grade;
   struct person descrip;
   char lunch[25];
   }   student[53];
struct alldat teacher,sub;
teacher.grade = 94;
teacher.descrip.age = 34;
teacher.descrip.status = 'M';
strcpy(teacher.decrip.name,"Mary Smith");
strcpy(teacher.lunch,"Baloney Sandwich");
sub.descrip.age = 87;
sub.descrip.status = 'M';
strcpy(sub.descrip.name,"Old Lady Brown");
sub.grade = 73;
strcpy(sub.lunch,"Yogurt and toast");
student[1][descrip]age = 15;
student[1][descrip]status = 'S';
strcpy(student[1][descrip]name,"Billy Boston");
strcpy(student[1]lunch,"Peanut Butter");
student[1][grade] = 77;
student[7][descrip]age = 14;
student[12][grade] = 87;
}
```

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 nearly 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 reserved word "struct".

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[25]", 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 we define an array of 53 variables each with the structure defined by "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.

11.11 Two More Variables

Since we have a new type definition we can use it to define two more variables. The variables "teacher" and "sub" are defined in the next statement to be variables of the type "alldat", so that each of these two variables contain 5 fields which can store data.

11.12 Now To Use Some Of The Fields

In the next five lines 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 whichis 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 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 "strcpy" function 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.

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 it's own internal stack to store the automatic variables on but most C compilers use only a small stack (typically 2048 bytes) as a default. This program has more than that in the defined structures so it will be necessary for you to increase the stack size. The method for doing this for some MS-DOS compilers is given in the accompanying COMPILER.DOC file with this tutorial. Consult your compiler documentation for details about your compiler, however this doesn't seem to be present in the HiTech compiler. Use the Applix `chmem` program to adjust the stack size of the executable code after compiling.

There is another way around this problem, and that is to move the structure definitions outside of the program where they will be external variables and therefore static. The result is that they will not be kept on the internal stack and the stack will therefore not overflow. It would be good for you to try both methods of fixing this problem.

11.13 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 will 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 tryingto 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.

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 "STRUCT.H".

11.14 What Are Unions?

Load 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. Examine the program on your monitor.

```
main( )
{
union {
   short int value;       /* This is the first part of the union  */
                      /* This has to be a short with the HI-TECH  */
                      /* C Compiler due to the int being 32 bits  */
   struct {
       char  first;   /* These two values are the second       */
```

```
        char second;
    } half;
} number;
long index;
    for (index = 12;index < 300000;index += 35231) {
        number.value = index;
        printf("%8x %6x %6x\n",number.value,number[half]first,
                number[half]second);
    }
}
```

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 most compilers, the data will be displayed with two 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 also.

When using the HiTech C compiler, data will be displayed with more than two leading 'f's, due to the use of the short int value.

Compile and run this program and observe that the data is read out as an "int" and as two "char" variables. The "char" variables are reversed in order because of the way an "int" variable is stored internally in your computer. Don't worry about this. It is not a problem but it can be a very interesting area of study if you are so inclined.

The 'char' variables are in their correct position when compiled with HiTech C.

11.15 Another Union Example

Load and display 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.

```
#define AUTO 1
#define BOAT 2
#define PLANE 3
#define SHIP 4
```

```
main( )
{
struct automobile {  /* structure for an automobile        */
   int tires;
   int fenders;
   int doors;
};
typedef struct {  /* structure for a boat or ship       */
   int displacement;
   char length;

}  BOATDEF;
struct {
   char vehicle;  /* what type of vehicle    */
   int weight;    /* gross weight of vehicle */
   union {        /* type-dependent data     */
       struct automobile car;     /* part 1 of the union     */
       BOATDEF boat;          /* part 2 of the union     */
       struct {
           char engines;
           int wingspan;
       } airplane;            /* part 3 of the union     */
       BOATDEF ship;          /* part 4 of the union     */
       } vehicle_type;
       int value;     /* value of vehicle in dollars    */
       char owner[32];    /* owners name */
} ford, sun_fish, piper_cub; /* three variable structures  */
       /* Define a few of the fields as an illustration     */
   ford.vehicle = AUTO;
   ford.weight = 2742;        /* with a full gas tank    */
   ford.vehicle_type.car.tires = 5;  /* including the spares    */
   ford.vehicle_type.car.doors = 2;
   sun_fish.value = 3742;         /* trailer not included    */
   sun_fish.vehicle_type.boat.length = 20;
   piper_cub.vehicle = PLANE;
   piper_cub.vehicle_type.airplane.wingspan = 27;
   if (ford.vehicle == AUTO) /* which it is in this case    */
       printf("?The ford has %d tires./n",ford.vechicle_type.car.tires);
   if (piper_cub.vehicle == AUTO) /* which it is not in this case  */
       printf("The plane has %d tires.\n",piper_cub.vechicle_type.
              car.tires);
}
```

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.

11.16 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 definition. Capitalizing the name is a personal preference onlyand is not a C standard. 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 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.

11.17 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 "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 to be stored in the variable "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.

11.18 Programming Exercises

1. Define a named structure containing a string field 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. etc.

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

12
Dynamic Allocation

12.1 What Is Dynamic Allocation?

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 simply variables that do not exist when the program is loaded, but are created dynamically as they are needed. It is possible, using these techniques, to create as many variables as needed, use them, and deallocate their space for use by other variables. As usual, the best teacher is an example, so load and display the program named `dynlist.c`.

```
main( )
{
struct animal {
   char name[25];
   char breed[25];
   int age;
   } *pet1, *pet2, *pet3;
   pet1 = (struct animal *)malloc(sizeof(struct animal));
   strcpy(pet1->name,"General");
   strcpy(pet1->breed,"Mixed Breed");
   pet1->age = 1;
   pet2 = pet1;   /* pet 2 now points to the above data structure  */
   pet1 = (struct animal *)malloc(sizeof(struct animal));
   strcpy(pet1->name,"Frank");
   strcpy(pet1->breed,"Labrador Retriever");
   pet1->age = 3;
   pet3 = (struct animal *)malloc(sizeof(struct animal));
   strcpy(pet3->name,Krystal");
   strcpy(pet3->breed,German Shepard");
   pet3->age = 4;
       /*   now print out the data described above  /*
   printf("%s is a %s, and is %d years old.\n",pet1->name,pet1->breed,
          pet1->age);
   printf("%s is a %s, and is %d years old.\n",pet2->name,pet2->breed,
          pet2->age);
   printf("%s is a %s and is %d years old.\n",pet3->name,pet3->breed,
          pet3->age);
   pet1 = pet3;   /* pet1 now points to the same structure that
                  pet3 points to                          */
   free(pet3);    /* this frees up one structure              */
   free(pet2);    /* this frees up one more structure         */
/*   free(pet1);  this cannot be done,see explanation in text     */
}
```

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 point to the defined structure. In order to do anything, we need some variables, so we will create some dynamically.

12.2 Dynamic Variable Creation

The first program statement, 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 other things 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 a "heap".

12.3 What Is A Heap?

Every compiler has a set of limitations on it as to how big the executable file can be, how many variables can be used, how long the source file can be, etc. In the 1616, the major limit is your total memory of 512k (or 4.5 megabyte, or whatever).

One limitation placed on users by many compilers for the IBM-PC and compatibles is a limit of 64K for the executable code. This is because the IBM-PC uses a microprocessor with a 64K segment size, and it requires special calls to use data outside of a single segment. In order to keep the program small and efficient, these calls are not used, and the size is limited but still adequate for most programs. This limitation does not apply to 1616/OS users, as the Motorola 68000 has a flat memory space of 16 megabyte.

A heap is an area which can be accessed by the program to store data and variables. The data and variables are put 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. Refer to your 1616/OS *Programmers Manual* for more details of memory allocation in the 1616.

12.4 More About Segments

This section applies only to MS-DOS users, and is a consequence of the brain damaged segmented design of the Intel 8086 processor. The design was forced upon Intel by a commercial requirement that they remain semi-compatible with their first microprocessors, the 8008 and 8080. This segmentation forces a choice of memory models on MS-DOS users, which their compilers try to make somewhat easier. This limitation was only overcome with the release of Intel 80386 and later processors, which can have large segments.

Someof the more expensive compilers givethe user a choice of memory models to use. Examples are Lattice and Microsoft, which allow the programmer a choice of using a model with a 64K limitation on program size but more efficient running, or using a model with a 640K limitation and requiring longer address calls leading to less efficient addressing. Using the larger address space requires inter segment addressing resulting in the slightly slower running time. The time is probably insignificant in most programs, but there are other considerations.

If an MS-DOS program uses no more than 64K bytes for the total of its code and memory and if it doesn't use a stack, it can be made into a `.com` file. Since a `.com` file is already in a memory image format, it can be loaded very quickly whereas a file in a `.exe` format must haveits addresses relocated as it is loaded. Therefore a small memory model can generate a programthat loads faster than one generated with a larger memory model. Don't let this worry you, it is a fine point that few programmers worry about.

Using dynamic allocation, it is possible to store the data on the "heap" and that may be enough to allow you to use the small memory model. Of course, you wouldn't store local variables such as counters and indexes on the heap, only very large arrays or structures.

Even more important than the need to stay within the small memory model is the need to stay within the computer. If you had a program that used several large data storage areas, but not at the same time, you could load one block storing it dynamically, then get rid of it and reuse the space for the next large block of data. Dynamically storing each block of data in succession, and using the same storage for each block may allow you to run your entire program in the computer without breaking it up into smaller programs.

12.5 Back To The "Malloc" Function

Hopefully the above 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 gets the block with the pointer 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" is a new function, new to us at least, that 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 sent to the system with the "malloc" call. At the completion of that call, we have a block on the heap allocated to us, with pet1 pointing to the first byte of the block.

12.6 What Is A Cast?

We still have a funny looking construct at the beginning of the "malloc" function call. That is called a "cast". The "malloc" function returns a block with the pointer pointing to it being a pointer of type "char" by default. Many times, if not most, you do not want a pointer to a "char" type variable, but 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 strange looking construct. 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 the warning message.

12.7 Using The Dynamically Allocated Memory Block

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 the next three lines of the program, we assign some silly data to the structure for illustration. It should comeas nosurprise toyouthatthese assignmentstatementslookjust likeassignments to statically defined variables.

In the next statement, we assign the value of "pet1" to "pet2" also. 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.

Finally, we allocate another block on the heap using the pointer "pet3", and fill its block with illustrative data.

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

12.8 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 block 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 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 disposed of. 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, and the second "free" call removes 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. 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, when we return to the OS, the entire heap will be disposed of with no regard to what we have put on it. The point does need to made that losing a pointer to a block of the heap, forever removes that block of data storage from our program and we may need that storage later.

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

12.9 That Was A Lot Of Discussion

It took nearly four 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 last four pages 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 files to study. But the fact remains, there is nothing more to learn about dynamic allocation than what was given so far in this chapter.

12.10 An Array Of Pointers

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".

```
main( )
{
struct animal {
    char  name[25];
    char  breed[25];
    int age;
} *pet[12],  *point;          /*this defines 13 pointers,no variables */
int index;

   /* first, fill the dynamic structures with nonsense */
    for (index = 0;index < 12;index++) {
       pet[index] = (struct animal *)malloc(sizeof(struct animal));
       strcpy(pet[index]->name,"General");
       strcpy(pet[index]->breed,"Mixed Breed");
       pet[index]->age = 4;
    }
```

```
    pet[4]->age = 12;            /*  these  lines  are  simply  to  */
    pet[5]->age = 15;            /*  put some nonsense data into  */
    pet[6]->age = 10;            /*  a few of the fields.  */
         /* now prints out the data described above */
    for (index = 0;index <12;index++) {
       point = pet[index];
       printf("%s is a %s, and is %d years old.\n", point->name,
              point->breed, point->age);
}
              /* good programming practice dictates that we free up the  */
              /* dynamically  allocated space  before we quit.           */
    for (index = 0;index < 12;index++)
       free(pet[index]);
}
```

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 pointer to a pointer. This is valid in C, and in fact you can go farther if needed but you will get quickly confused. I know of no limit as to how many levels of pointing are possible, so a definition such as "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 in a database, readings from some test equipment, or any other source of data.

A few fields are randomly picked 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. As stated earlier, there was nothing new here about dynamic allocation, only about an array of pointers.

12.11 A Linked List

We finally come to the grandaddy 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.

```
#include "stdio.h" /* this is needed only to define the NULL */
#define RECORDS 6

main( )
{
struct animal {
   char name[25];                    /* The animals name       */
   char breed[25];                   /* The type of animal     */
   int age;                          /* The animals age        */
   struct animal *next;      /* a pointer to another record of this type */
} *point, *start, *prior;     /* this defines 3 pointers, no variables */
int index;

                        /* the first record is always a special case */
```

```
    start = (struct animal *)malloc(sizeof(struct animal));
    strcpy(start ->name,"general");
    strcpy(start ->breed,"Mixed Breed");
    start->next = NULL;
    prior = start;
        /* a loop can be used to fill in the rest once it is started    */
    for (index = 0;index < RECORDS;index++) {
        point = (struct animal *)malloc(sizeof(struct animal));
        strcpy(point->name,"Frank");
        strcpy(point->breed,"Laborador Retriever");
        point->age = 3;
        point->next = point /* point last "next" to this record */
        point->next = NULL;   /* point this "next" to NULL     */ prior
        = point;  /* this is now the prior record  */
    }
            /* now print out the data described above  */
    point = start;
    do {
        prior = point->next;
        printf("%s is a %s,and is %d years old.\n", point->name,
                point->breed, point->age);
        point = point->next;
    } while (prior  != NULL);
        /* good programming practice dictates that we free up the   */
        /* dynamically allocated space before we quit           */
    point = start;              /* first block of group         */
    do {
        prior  =  point->next;  /*  next  block  of  data  */
        free(point);          /* free present block    */point
        = prior;         /* point to next                    */
       } while (prior != NULL);   /* quit when next is NULL     */
}
```

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 as your sister forced you to do. We will however, 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 retraverse the entire list again and again if we so desire.

12.12 The Data Definitions

This program starts similarly to the last two with the addition of the definition of a constant 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, 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.

12.13 The First Field

Using the "malloc" function, we request a block of storage on the "heap" and fill it with data. The additional field in this example, the pointer, is assigned the value of NULL, which is only used to indicate that this is the end of the list. We will leave the pointer "start" 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.

12.14 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 our program. Each time through, we allocate memory, 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.

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.

1. "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 that points to NULL and can be used to detect the end as shown below.

start->struct1 name breed age point->struct2 name breed age
point->struct3 name breed age point-> ..... struct7 name breed age
point->NULL

It should be clear to you, if you understand the above structure, that it is not possible to simply jump into the middle of the structure 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 structure 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.

12.15 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 on 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. Caremust 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.

12.16 More About Dynamic Allocation And Linked Lists

It is not difficult, and it is not trivial, to add elements into the middle of a linked lists. 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.

12.17 Another New Function - Calloc

One more function must be mentioned, the "calloc" function. This 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.

12.18 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.

13
Character and Bit Manipulation

13.1 Upper And Lower Case

Load and display the program `uplow.c` for an example of a program that does lots of character manipulation. For a description of the `stdio.h` file, see the `dosex_1616.c` program. More specifically, `uplow.c` changes the case of alphabetic characters around. 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_line". Compile and run the program with the file of your choice. The four functions are;

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

```
#include "/sys/stdio.h"
#include"/sys/ctype.h"    /* Note - your compiler may not need this    */

main( )
{
FILE *fp;
char line[80], filename[24];
char *c;

   printf("Enter filename -> ");
   scanf("%s",filename);
   fp = fopen(filename,"r");

   do {
       c = fgets(line,80,fp);     /* get a line of text  */
       if (c  != NULL) {
           mix_up_the_chars(line);
       }
   } while (c != NULL);

   fclose(fp);
}

mix_up_the_chars(line)    /*      this function turns all upper case
                      characters into lower case, and all
                      lower case into upper case. It ignores
                      all other characters.             */

char line[];
{
int index;

   for (index = 0;line[index] != 0;index++) {
       if (isupper(line[index]))     /* 1 if upper case     */
           line[index] = tolower(line[index]);
       else {
           if (islower(line[index])) /* 1 if lower case     */
           line[index] = toupper(line[index]);
       }
   }
   printf("%s",line);
}
```

13.2 Classification Of Characters

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. There are several others that are commonly used, so they are defined in the following table;

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

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.

The program on your screen uses 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(); Is the character alphabetic?
isdigit(); Is the character a numeral?
isspace(); Is the character any of, \n, \t, or blank?

```
#include "/sys/stdio.h"
#include "/sys/ctype.h"     /*note your compiler may not need this  */

main( )
{
FILE *fp;
char line[80],filename[24];
char *c;

   printf("Enter filename -> ");
   scanf("%s",filename);
   fp = fopen(filename,"r");

   do  {
       c = fgets(line,80,fp);   /* get a line of text  */
           if (c != NULL) {
           count_the_data(line);
           }
       } while (c != NULL);

   fclose(fp);
}

count_the_data(line)
char line[];
{
int whites, chars, digits;
int index;

   whites = chars = digits = 0;

   for (index = 0;line[index] != 0;index++) {
       if (isalpha(line[index]))    /* 1 if line.. is alphabetic   */
           chars++;
       if (isdigit(line[index]))    /* 1 if line.. is a digit      */
           digits++;
       if (isspace(line[index]))    /* 1 if line.. is blank, tab,  */
           whites++;               /*         or newline          */
   }    /*end of counting loop   */

   printf("%3d%3d%3d %s",whites,chars,digits,line);
}
```

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. See also `dosex1616.c` in Chapter 14 for another example.

13.3 The Logical Functions

Load and display the program `bitops.c`.

```
main( )
{
char mask;
char number.6,;
char  and,or,xor,inv,index;
   number[0]  =  0X00;
   number[1]  =  0X11;
   number[2]  =  0X22;
   number[3]  =  0X44;
   number[4]  =  0X88;
   number[5] =  0Xff;
   printf("nmbr  mask  and or xor inv\n");
   mask =  0X0F;
   for (index = 0;index <= 5;index++) {
       and = mask & number[index];
       or = mask l number[index];
       xor = mask ^ number[index];
       inv = ~number[index];
       printf("%5x %5x %5x %5x %5x %5x\n",number.index.,
             mask,and,or,xor,inv);
   }
   printf("\n");
   mask = 0X22;
   for (index = 0;index <= 5;index++) {
       and = mask & number[index];
       or = mask l number[index];
       xor = mask ^ number[index];
       inv = ~number[index];
       printf("%5x %5x %5x %5x %5x %5x\n",number[index],
           mask,and,or,xor,inv);
   }
}
```

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 the bit is 1, the result is 0, and if the 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.

Run the program and observe the output.

13.4 The Shift Instructions

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.

```
main( )
{
int small,big,index,count;
   printf("      shift left       shift right\n\n");
   small = 1;
   big = 0x4000;
   for(index = 0;index < 17;index++) {
       printf("%8d %8x %8x\n",small,small,big);
       small = small << 1;
       big = big >> 1;
   }

   printf("\n");
   count = 2;
   small = 1;
   big = 0x4000;
   for(index = 0;index < 9;index++) {
       printf("%8d %8x %8x\n",small,small,big);
       small = small << count;
       big = big >> count;
   }
}
```

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.

14
Example Programs

14.1 Why This Chapter?

Although every program in this tutorial has been a complete program, each one has also been a very small program intended to teach you some principle of programming in C. It would do you a disservice to leave you at that point without introducing you to a few larger programs to illustrate how to put together the constructs you have learned to create a major program. This chapter contains four programs of increasing complexity, each designed to take you into a higher plateau of programming, and each designed to be useful to you in some way.

DOSEX was intended to illustrate how to make MS-DOS system calls and will teach you, through self-study, how the system responds to the keyboard. DOSEX has been modified by Tim Ward to use Applix 1616 system calls instead of MS-DOS calls.

WHATNEXT reads commands input on the command line and will aid you in setting up a variable batch file, one that requests an operator input and responds to the input by branching to a different part of the batch file.

LIST is the source code for the program you used to print out the C source files when you began studying C with the aid of this tutorial. Finally we come to VC, the Visual Calculator, which you should find to be a useful program even if you don't study its source code. VC uses most of the programming techniques we have studied in this course and a few that we never even mentioned such as separately compiled subroutines.

We will take a look at the example programs one at a time but without a complete explanation of any of them because you have been studying C for some time now and should be able to read and understand most of these programs on your own. One other thing must be mentioned, these programs use lots of nonstandard constructs and you will probably need to modify some of them to get them to compile with your particular compiler. That will be left as an exercise for you.

`dosexibm.c` - The MS-DOS Example Program

The copy of MS-DOS that you received with your IBM-PC or compatible has many internal MS-DOS calls that you can use as a programmer to control your peripheral devices and read information or status from them. Some of the earlier IBM DOS manuals, MS-DOS 2.0 and earlier, have these calls listed in the back of the manual along with how to use them. Most of the manuals supplied with compatible computers make no mention of these calls even though they are extremely useful. These calls can be accessed from nearly any programming language but they do require some initial study to learn how to use them. This program is intended to aid you in this study.

Display the program on your monitor or print it out for reference. It is merely a loop watching for a keyboard input or a change in the time. If either happens, it reacts accordingly. In line 23, the function "kbhit()" returns a value of 1 if a key has been hit but not yet read from the input buffer by the program. This is a nonstandard function and may require a name change foryour particular compiler. There will probably be several similar calls that will need changed for your compiler in order to compile and run the programs in chapter 14.

Look at the function named "get_time" for an example of a MS-DOS call. An interrupt 21(hex) is called after setting the AH register to 2C(hex) = 44(decimal). The time is returned in the CH, CL, and DH registers. Refer to the MS-DOS call definitions in your copy of MS-DOS (if available). If the definitions are not included there, Peter Norton's book, "Programmers Guide

to the IBM PC" is recommended as a good reference manual for these calls and many other programming techniques. You will notice it is somewhat more complex than the Applix 1616 equivalent.

Another useful function is the "pos_cursor()" function that positions the cursor anywhere on the monitor that you desire by using a MS-DOS interrupt. This call has been modified in the Applix 1616 version, to use an equivalent system call. In the MS-DOS case, the interrupt used is 10(hex) which is the general monitor interrupt. This particular service is number 2 of about 10 different monitor services available. This particular function may not be needed by your compiler because some compilers have a cursor positioning function predefined for your use. This function is included here as another example to you.

The next function, service number 6 of interrupt 10(hex) is the window scroll service. It should be self explanatory. The window scroll service has been deleted from the Applix version. Full details of Conal Walsh's window support are provided in the tutorials commencing in µ*Peripheral* issue 10.

In this program, the cursor is positioned and some data is output to the monitor, then the cursor is "hidden" by moving it to line 26 which is not displayed. After you compile and run the program, you will notice that the cursor is not visible on the monitor. This is possible in any program, but be sure to put the cursor in view before returning to DOS because DOS does not like to have a "hidden" cursor and may do some strange things.

Some time spent studying this program will be valuable to you as it will reveal how the keyboard data is input to the computer. Especially of importance is how the special keys such as function keys, arrows, etc. are handled.

`dosex_1616.c` - is the Applix 1616 program equivalent to the above.

```
/* ****************************************************************/
/* This is an example program to illustrate how to ;                */
/* 1. Get the time and date from APPLIX 1616/OS using a system call     */
/* 2. Set the cursor to any position on the screen using a 1616/OS video*/
/* driver escape code                                               */
/* 3. Read characters from the keyboard and display their codes         */
/* 4. How to scroll a window up on the monitor                      */
/* 5. Format a program for ease of reading and understanding            */
/* ****************************************************************/
#include "/sys/stdio.h"
#include "/sys/syscalls.h"
#include "/sys/ctype.h"
main( )
{
int character, x;
x = 2;
   draw_box();            /* draw the boxes around the fields */
   pos_cursor(33,19);
   printf("Type Q to quit");
   do {
       if  (kbhit()) {              /* has a key been hit  */
           if (x <= 16) {         /* scroll output in window down a */
               pos_cursor(9,x+1);  /* line.                    */
               ++x;
           }
           else if
               (x <= 31){
               pos_cursor(50,x-14); /* return to top of window on right */
               ++x;                 /* hand side.                */
           }
           else {
               x = 2;               /* return to top of window on left  */
               pos_cursor(9,x+1);   /* hand side.                    */
           }
           character = getch();     /* read it in           */
```

```
            disp_char(character);     /* display it         */
        }
        get_time();    /* get the time of day and display it.    */

    }   while (character != 'Q'); /* Quit when a Q is found */

    pos_cursor(0,0);          /* put cursor at top of screen */
}
/* ********************************************************** drawbox */
/* This routine draws a box on the screen. The keys hit, the time and  */
/* date are displayed in these boxes.  Nothing special about these     */
/* boxes, they are simple output using the printf function.           */
/* ********************************************************************* */
draw_box()
{
int index;
char line[81];
    for (index = 0;index < 80;index++)       /* three blank rows */
        line[index] = ' ';
    line[80] = NULL;                  /* end of string   */
    for (index = 0;index < 3;index++)
        printf("%s",line);
    line[8] = 201;                /* draw top line of box    */
    for (index = 9;index < 70;index++)
        line[index] = 205;
    line[70] = 187;
    printf("%s",line);
    line[8] = 186;            /* draw sides of large box    */
    for (index = 9;index < 70;index++)
        line[index] = ' ';
    line[70] = 186;
    for (index = 0;index <15;index++)
        printf("%s",line);
    line[8] = 204;            /*   draw line between boxes */
    for (index = 9;index < 70;index++)
        line[index] = 205;
    line[70] = 185;
    printf("%s",line);
    line[8] = 186;            /*   sides of time/date box   */
    for (index = 9;index < 70;index++)
        line[index] = ' ';
    line[70] = 186;
    printf("%s",line);
    line[8] = 200;            /* bottom line of the box  */
    for (index = 9;index < 70;index++)
        line[index] = 205;
    line[70] = 188;
    printf("%s",line);
    for (index = 0;index < 80;index++)       /* three blank rows    */
        line[index] = ' ';
    for (index = 0;index < 3;index++)
        printf("%s",line);
}
/* ********************************************************** disp_char */
/* This routine displays characters hit on the monitor. If the first   */
/* character is a zero, it is a special character hit, and the zero is */
/* displayed. Next character is read, and is displayed on the monitor. */
/* ********************************************************************* */
disp_char(inchar)
int inchar;
{
int c;
    c = inchar;
```

```
       if (c == 13) {
           return;
       }
       if ( isupper(c) ) {
           printf("%4d (%c) Uppercase  ",c,c);
       }
       if ( islower(c) ) {
           printf("%4d (%c) Lowercase  ",c,c);
       }
       if ( isspace(c) ) {
           printf("%4d (%c) White Space",c,c);
       }
       if ( ispunct(c) ) {
           printf("%4d (%c) Punctuation",c,c);
}
       if ( isdigit(c) ) {
           printf("%4d (%c) Digit      ",c,c);
       }
   pos_cursor(25,1);  /* hide the cursor on the 26th line   */
}
/* ********************************************************* pos_cursor */
/* Function is used to position the cursor on the screen using the    */
/* 1616OS video escape call                                          */
/* ***************************************************************** */
pos_cursor(xpos,ypos)
int  xpos, ypos;
{
   printf("\033=%c%c", ypos + 32, xpos + 32);
}
/* ********************************************************get_time */
/* Routine calls the 1616OS function call for time of day.  Returns */
/* the time of day to calling program in a do while loop so it      */
/* is conviently updated                                          */
/* ***************************************************************** */
get_time()
{
char *time, BUF[22];
   time = GETTDSTR(BUF);      /* 1616 syscall for timedate string  */
   pos_cursor(10,19);         /* cursor positioning function */
   printf("%s",time);
   pos_cursor(50,19);
   printf("%s",time);
}
```

14.2 `whatnext.c` - The Batch File Interrogator

This is an example of how to read the data on the command line following the function call. Notice that there are two variables listed within the parentheses following the main() call. The first variable is a count of words in the entire command line including the command itself and the second variable is a pointer to an array of pointers defining the actual words on the command line.

First the question on the command line, made up of some number of words, is displayed on the monitor and the program waits for the operator to hit a key. If the key hit is one of those in the last "word" of the group of words on the command line, the number of the character within the group is returned to the program where it can be tested with the "errorlevel" command in the batch file. You could use this technique to create a variable AUTOEXEC.BAT file or any other batch file can use this for a many way branch. Compile and run this file with TEST.BAT for an example of how it works in practice. You may find this technique useful in one of your batch files and you will almost certainly need to read in the command line parameters someday. Since

1616/OS doesn't use batch files like MS-DOS ones, this program isn't all that much use, but it is left in as an example of how to pass parameters from a command line, and use them in a program.

An interesting alternative would be for you to write a program named "would.c" that would return a 1 if a "Y" or "y" were typed and a zero if any other key were hit. Then your batch file could have a line including such a test.

```
/* ************************************************************** */
/* This program reads a series of words from the command line,and
*/
/* displays all but the last on the monitor. The last is a series of   */
/* characters which are used as input comparisons. One character is    */
/* read from the keyboard. If it is one of the characters in the       */
/* comparision list, its number is returned to DOS as the errorlevel   */
/* command. If the character does not exist in the list, a zero is
*/
/* returned. Example follows;                                          */
/*                                                                     */
/* WHATNEXT What model do you want? ALR%3T                             */
/*     If key a or A is hit,errorlevel 1 is returned.                  */
/*     If key l or L is hit,errorlevel 2 is returned.                  */
/*     If key r or R is hit,errorlevel 3 is returned.                  */
/*     If key % is hit, errorlevel 4 is returned.                      */
/*     If key 3 is hit, errorlevel 5 is returned.                      */
/*     If key t or T is hit,errorlevel 6 is returned.                  */
/*     If any other key is hit, errorlevel 0 is returned.              */
/*                                                              */
/* The question must be on one line.                               */
/* Up to nine different keys can be used.                          */
/* The errorlevel can be interpreted in a batchfile.               */
/* ************************************************************** */

#include "/sys/stdio.h"
#include "/sys/ctype.h"

main(number,name)
int number;               /* total number of words on command line */
char *name[];
{
int index;                /* a counter and incrementing variable   */
int c;                    /* the character read in for comparison */
int code;              /* the resulting errorlevel returned to */
char next_char;        /* used for the comparison loop   */
char *point;           /* a dummy pointer used for convenience   */

                  /* At least one group must be used for this  */
                  /* filename,and one group used for the       */
                  /* required fields, so less than three allows */
                  /* for no questions.                  */
   if (number < 3) {
   printf("No question given on command line\n");
   exit(0);
}
                  /* print out words 2 to n-1, the question    */
   number--;

   for(index = 1;index < number;index++) {
       printf("%s ",name[index]);
   }
                  /* get the users response and make it uppercase */
   c = getch();
   printf("%c\n",c);
   if (islower(c))
       c = toupper(c);
   point = name[number]; /* point to the last pointer on the inputs    */
```

```
    code = 0;
    index = 0;
    do {            /* search across allowed responses in last word  */
        next_char = *(point + index);
        if (islower(next_char))
            next_char = toupper(next_char);   /* make it uppercase */
        if(next_char == c)                    /* if a match is found */
            code = index + 1;      /* save the number of the match */
        index++;
    } while (*(point + index));       /* until NULL terminator found */
    exit(code);               /* return the errorcode to the system */
}
```

14.3 `List.C` - The Program Lister

This program is actually composed of two files, `list.c` and `listf.c` that must be separately compiled and linked together with your linker. There is nothing new here and you should have no trouble compiling and linking this program by reading the documentation supplied with your compiler.

The way to compile them with HiTech C is:

`relcc -v -W1 list.c listf.c` Enter

```
/* ************************************************************** */
/* This program will read in any file and list it on the monitor with  */
/* line numbers and with page numbers.                              */
/* ************************************************************** */
#include "/sys/stdio.h"      /* standard I/O header file   */
#define MAXCHARS 255         /* maximum size of a line */
FILE *file_point;               /* point to file to be read */
FILE *print_file_point;      /* pointer to pronter  */
extern top_of_page();
char oneline[256];              /* input string buffer area */

main(number,name)
int number;           /* number of arguments on command line    */
char *name[];         /* arguments on the command line      */
{
char *c;              /* variable to indicate end of file   */
char *point;

   point = name[1];
   open_file(number,point);  /* open the file to read and print    */
   open_print_file();
   do {
       c = fgets(oneline,MAXCHARS,file_point);   /*  read one line  */
       if (c != NULL)
           print_a_line();                   /*   print a line  */
   } while (c !=NULL);                   /*   continue until EOF   */

   top_of_page();                /* move paper to top of page */
   close(file_point);                    /* close read file     */
   close(print_file_point);              /*  close printer file     */
}
```

The only thing that is new in this program is the inclusion of three "extern" variables in the `listf.c` listing. The only purpose for this is to tie these global variables to the main program and tell the compiler that these are not new variables. The compiler will therefore not generate any new storage space for them but simply use their names during the compile process. At link time, the linker will get their actual storage locations from the LIST.OBJ file and use those locations for the variables in the LISTF part of the memory map also. The variables of those names in both files are therefore the same identical variables and can be used just as any other global variables could be used if both parts of the program were in one file.

```
/* ********************************************************************** */
/* Module contains the functions called by the list.c program. If this  */
/* were a program to be used for  specific purpose,it would probably not*/
/* be wise to break it into two separately compiled modules.It is only  */
/* done here for purposes of illustration. It is a useful program.
*/
/* ********************************************************************** */
#define MAXLINES 54          /*     maximum number of lines per page     */
#include "/sys/stdio.h"              /*   standard I/O header file   */
extern FILE *file_point;         /*    point to the file to be read   */
extern FILE *print_file_point;       /*    pointer to the printer    */
extern char oneline[];               /*   input string buffer area   */
char filename[15];               /*   filename from header or prompt  */
int line_number = 0;         /*   line number initialized to one  */
int page_number = 1;         /*   page number initialized to one  */
int lines_this_page = 0;             /*   lines on this page so far  */

/* ********************************************************* open_file */
/* This functions opens input file on the command line,if there was  */
/* one defined. Otherwise, requests a file to open and opens the     */
/* requested file.                                                   */
/* ********************************************************************** */
open_file(no,name)
int no;                  /*   number of arguments on command line  */
char *name;                  /*  first argument from command line */
{
   strcpy(filename,name);        /*   copy name for printing header   */
   file_point = NULL;        /*    if no name was given in command      */
   if (no == 2) {        /*   2nd field in command is filename      */
       file_point = fopen(name,"r");     /*   open requested file   */
       if (file_point == NULL)       /*  NULL if file doesn't exist  */
           printf("File name on command line doesn't exist!\n");
   }

     do {
       if (file_point == NULL) {         /*   no filename yet  */
       printf("Enter filename -> ");
       scanf("%s",filename);
       file_point = fopen(filename,"r");                /* open file */
       if (file_point == NULL)           /*   NULL if file no exist */
       printf("Filename doesn't exist,try again.\n");
   }
  } while (file_point == NULL);      /* continue until good filename  */
}
/* ****************************************************** open_print_file
*/
/* This function opens the printer file to the standard printer.
*/
/* ********************************************************************
*/
open_print_file()
{
   print_file_point = fopen("PRN","w");      /* open printer file   */
}
/* ******************************************************** print_a_line
*/
/* This routine prints a text line and checks to see if there is room for
*/
/* another on the page. If not,it starts a new page with a new header.
*/
/* This routine calls several other local routines.
*/
/* ********************************************************************
*/
```

```
print_a_line()
{
int index;
   header();
   printf("%5d %s",line_number,oneline);
                /*   This prints a line of less than 72 chars */
   if (strlen(oneline) < 72)
       fprintf(print_file_point,"%5d %s",line_number,oneline);
                /*   This prints a line of 72 to 143 chars   */
   else if (strlen(oneline) < 144) {
       fprintf(print_file_point,"%5d ",line_number);
       for (index = 0;index < 72;index++)
           fprintf(print_file_point,"%c",oneline[index]);
       fprintf(print_file_point,"<\n      ");
       for (index = 72;index < strlen(oneline);index++)
           fprintf(print_file_point,"%c",oneline[index]);
       lines_this_page++;
   }
                /*   This prints a line of 144 to 235 chars  */
       else if (strlen(oneline) < 235) {
           fprintf(print_file_point,"%5d ",line_number);
       for (index = 0;index < 72;index++)
           fprintf(print_file_point,"%c",oneline[index]);
       fprintf(print_file_point,"<\n      ");
       for (index = 72;index < 144;index++)
           fprintf(print_file_point,"%c",oneline[index]);
       fprintf(print_file_point,"<\n      ");
       for (index = 144;index < strlen(oneline);index++)
           fprintf(print_file_point,"%c",oneline[index]);
       lines_this_page += 2;
   }
       /* the following line outputs a newline if there is none
                                  at the end of the last line   */
   if (oneline[strlen(oneline)-1] != '\n')
       fprintf(print_file_point,"%c",'\n');
   line_number++;
   lines_this_page++;
}
/* ********************************************************   header */
/* This routine checks if a header needs to be printed.It also checks */
/* for the end of a page,and spaces the paper up.                     */
/* ****************************************************************** */
header()
{
int index;
          /*   first see if we are at the bottom of the page */
   if (lines_this_page > MAXLINES) {     /* space paper up from bottom */
   for (index = lines_this_page;index < 61;index++)
       fprintf(print_file_point,"\n");
   lines_this_page = 0;
   }
       /* put a monitor header out only at the very beginning  */
   if (line_number == 0) {           /* display monitor header */
       printf("       Source file %s\n",filename);
       line_number = 1;
   }
       /* check to see if we are at the top of the page either  */
       /* through starting a file,or following a bottom of page */
   if (lines_this_page == 0) {   /* top of every printer page */
       fprintf(print_file_point,"\n\n\n       ");
       fprintf(print_file_point," Source file - %s     ",filename);
       fprintf(print_file_point,"        Page %d\n\n",page_number);
```

```
        page_number++;
    }
}
/* ********************************************************* top_of_page  */
/* This function spaces the paper to the top of the next page so another */
/* call to this function will start correctly. Used only at the end      */
/* of a complete printout.                                               */
/* ********************************************************************* */
top_of_page()
{
int index;
    for (index = lines_this_page;index < 61;index++)
        fprintf(print_file_point,"\n");
}
```

There is no reason why the variables couldn't have been defined in the `listf.c` part of the program and declared as "extern" in the `list.c` part. Some of the variables could have been defined in one and some in the other. It is merely a matter of personal taste. Carried to an extreme, all of the variables could have been defined in a third file and named "extern" in both of these files. The third file would then be compiled and included in the linking process.

It would be to your advantage to compile, link, and run this program to prepare you for the next program which is composed of 5 separate files which must all work together.

14.4 `vc.c` - The Visual Calculator

This program finally ties nearly everything together because it uses nearly every concept covered in the entire tutorial. It is so big that I will not even try to cover the finer points of its operation. Only a few of the more important points will be discussed.

The first thing you should do is go through the tutorial for VC included in the file VC.DOC. There are several dozen steps for you to execute, with each step illustrating some aspect of the Visual Calculator. You will get a good feel for what it is capable of doing and make your study of the source code very profitable. In addition, you will probably find many ways to use the Visual Calculator to solve problems involving calculations where the simplicity of the problem at hand does not warrant writing a program.

Notice that the structure definitions, used in all of the separate parts of the program, are defined in the file STRUCT.DEF (renamed STRUCT.H in the Applix 1616 version). You should include STRUCT.H with the other Applix 1616 header files. During program development, when it became necessary to change one of the structures slightly, it was not necessary to change it in all of the files, only one file required modification which was then "included" in the source files. Notice that the transcript data is stored in a doubly linked list with the data itself being stored in a separate dynamically allocated char string. This line is pointed to by the pointer "lineloc".

For ease of development, the similar functions were grouped together and compiled separately. Thus, all of the functions involving the monitor were included in the file named `video.c`, and all of the functions involving the data storage were grouped into the `file.c` collection. Dividing your program in a way similar to this should simplify debugging and future modifications.

Of special interest is the "monitor()" function. In the Applix 1616 version, this function is deleted, as the 1616 supports both colour and monochrome monitors without change. In the MS-DOS world, this function examines the video mode through use of a DOS command and if it is a 7, it assumes it is a monochrome monitor, otherwise it assumes a color monitor. The colors of the various fields are established at this time and used throughout the program. Most of the data is written directly to the video memory, but some is written through the standard MS-DOS BIOS routines.

The file `definc.c` is simply a catalogue of the functions to aid in finding the functions. This file was generated as one of the first files and was maintained and updated for use during the entire design and coding lifetime.

Feel free, after understanding this code, to modify it in any way you desire for your own use.

To compile and link this, copy all five of the VC files (`data.c, file.c, vc.c, video.c, struct.h`) into a directory and include that directory in your `xpath`. Type:
`relcc -v -W1 -lf vc.c data.c video.c file.c`
(Note that the -lf option must be included so that the compiler uses the float library, while the others are optional.) Or use the `makefile` included on the distribution disk, if you have the `make` utility.

```
#include "/sys/ctype.h"
#include "/sys/stdio.h"
#include "/f0/struct.h"
#include "/sys/syscalls.h"
struct vars allvars[12];     /* this is the  main variable storage    */
int varinuse = 0;               /* which variable is being used now     */
char inline[60];             /* input line area                       */
int col;                     /* used for searching across the input   */
int errcode;                 /* error code number                  */
int colerr;                  /* column where error occured         */
int printit = 1;             /* 1 = print a transcript             */
int ignore;                  /* 1 = ignore calculations for line      */
extern char strngout[];      /* output message area                */
struct lines *top, *bot, *q, *p, *arrow, *trnsend;

/* ********************************************************** main */
/* The main control loop for the program. Initializes everything      */
/* and reads statements until no errors. Continues reading until      */
/* F10 occurs in subordinate function where control is returned to    */
/* 1616OS.                                                            */
main()
{
   top = bot = q = p = arrow = trnsend = NULL;
   initfk();                              /* sets up function keys */
   initdata(&allvars[0]);            /* initializes all data           */
   bkgndvid();            /* display video background - double lines   */
   valusvid();            /* display starting values of all variables */

    strtrans("Welcome to the Visual Calculator - Applix 1616 - Version
1.00",0);transout();
   do {
       poscurs(22,7);
       printf("  input >
");
       printf("             ");
   do {                            /* repeat untill no errors        */
       readline();                 /* get an input line          */
       errdis("                          ");  /*   clear error msg   */
       parse();                            /* parse the line */
if (errcode)
             errout();             /* output error message    */
   } while (errcode);
   if (ignore == 1)
       strtrans(inline,0);              /* store comment in transcript */
   else
       strtrans(inline,1);              /* store "inline" in transcript */
       transout();
   }   while (1);                                  /* continuous loop    */
}
/* ********************************************************** readline */
/* This function reads a line by inputting one character at a time and */
/* deciding what to do with it if a special character, or adding it to */
/* the input line if not a special character. The routine takes care of*/
/* such things as backspace, cursor movement and delete keys. The final*/
```

```
/* result is a line of text stored in the buffer "inline" and the line */
/* displayed on the monitor.                                           */
readline()
{
int index;
int c,temp;
int row = 22,col = 17;
   if (errcode) {                /* error recovery allow reenter   */
       index = colerr;
       errcode = 0;
   }
   else {                            /* normal input routine    */
       index = 0;
       for (temp = 0;temp < 60;temp++)
           inline[temp] = 0;             /* clear input buffer  */
   }
   poscurs(row,col+index);           /* starting location of cursor    */
   do {               /* repeat this do loop until a return is hit   */
       while ((c = getch()) == EOF);       /* get a keystroke        */if
       ((c == 2) || (c == 3) || (c == 4) || (c == 5) || (c == 12) ||
               (c == 24) || (c == 18) || (c == 19) || (c == 127)) {
           switch (c) {
                    case 19 : if (index) {               /* left arrow */
                       index = index -1;  /* back up cursor  */
                       }
                       break;
                   case 4 : if (index < 61) {            /* right arrow */
                           if (inline[index] == 0);   /* zero found */
                           inline[index] = ' ';   /* blank over 0 */
                           index = index + 1;         /* cursor forward */
                       }
                       break;
               case 5 : movarrow(-1);                /* up arrow    */
                       break;
               case 24 : movarrow(1);                /* down arrow */
                       break;

               case 18 : movarrow(-8);                   /* page up */
                       break;
                case 3  : movarrow(8);               /* page down */
                       break;
               case 12 : movarrow(-1000);                /* home */
                       break;

               case 2  : movarrow(1000);             /* end */
                       break;
                   case 127 : temp = index;              /* delete key  */
                           /* move all characters left one space    */
                       while (inline[temp]) {
                           inline[temp] = inline[temp+1];
                           putchar(inline[temp++]);
                       }
                       putchar(' ');      /* blank in last place    */
                       break;
               default : poscurs(15,5);
                       printf(" S%3d",c);
           }
           poscurs(row,col+index);   /* actually put cursor in position*/
       }
```

```
if (c == 0) {          /* a zero here says a special key was hit    */
                       /* get the key and act on it as needed   */
    int spec;
    spec = getch();        /* this is the special code found        */
    switch (spec) {
        case 59 : helpm();                    /* F1 - Help math  */
                transout();
                break;
        case 60 : helps();                    /* F2 - Help system   */
                transout();
                break;
        case 61 :
                break;             /* F3 - Spare */
        case 62 :                  /* F4 - Mark transcript   */
                arrow->marked = (arrow->marked?0:1);
                transout();
                break;
        case 63 : fileout();           /* F5 - Store transcript */
                break;
        case 64 : filein();               /* F6 - retrieve trans    */
                errcode = 0;
                break;
        case 65 :
                break;                 /* F7 - Spare */
        case 66 :
                break;                 /* F8 - Spare  */
        case 67 :                  /* F9 -   Edit a line  */
                strcpy(inline,arrow->lineloc);
                poscurs(22,17);
                printf("%s",inline);
                break;
        case 68 : poscurs(22,17);      /* F10 - Quit to 1616OS   */
                printf("Quit? (Y/N) ");
                c = getch();
                if ((c == 'Y') || (c == 'y')) {
                    clrscrn();
                    exit(0);
                }
                else {
                    poscurs(22,17);
                    printf("                 ");
                    break;
                }
        default : poscurs(15,5);
                printf(" S%3d",spec);
    }
    poscurs(row,col+index);   /* actually put cursor in position */
}
    else {                         /* normal letter or char hit */
    int curr, next;
    if (islower(c)) c = toupper(c);       /* convert to upper case */
    if ((c >= '\40') && (c <= '\176')) {         /* printable char  */
        poscurs(row,col+index);
        putchar(c);
        next = inline[index];
            inline[index++] = c;
        curr = index;
        while((next  != 0) && (curr <=60)) {  /* move remainder  */
            temp = next;                      /* line right  */
            next = inline[curr];
            inline[curr++] = temp;
            putchar(temp);
        }
    }
    else {
        if ((c == 8) && index){                  /* backspace   */
```

```
                index--;
                poscurs(row,col+index);           /* back up cursor  */
                temp = index;
                while (inline[temp]) {
                    inline[temp] = inline[temp+1];
                    putchar(inline[temp++]);
                }
                putchar(' ');
            }
        }
        poscurs(row,col+index);
      }
   } while (c  != 13);           /* newline found, line input is complete
*/
}
/* ********************************************************** parse  */
/* This function does checking of the input line for logical errors in */
/* construction, then turns control over to the function "calcdata" for*/
/* the actual calculations.                                            */
parse()
{
int index,parcol;
double newval;
char name[8];

   varinuse = -1;
   errcode = 0;
   col = 0;
   ignore = 1;                                /* ignore this line    */
   if (inline[0] == '#') {              /* get list of variable names */
       getnames();
       return;
   }
   while (inline[col] == ' ') col++;            /* ignore leading blanks */
   if (inline[col] == '$') return;              /* ignore a comment line */
   if (inline[col] == 0) return;             /* ignore a blank line */
   ignore = 0;                        /* don't ignore this line */
   name[0] = inline[col++];                   /* find variable name  */
   index = 1;
   while ((((inline[col] >= 'A' ) && (inline[col] <= 'Z' )) ||
         ((inline[col] >= '0' ) && (inline[col] <= '9' ))) &&
         (index <= 5 )) {            /* continue var or function name  */
       name[index++] = inline[col++];
   }
   name[index] = 0;
   for (index = 0;index < 12;index++) {
       if ((strcmp(name,allvars[index].varname)) == 0)
               varinuse = index;          /* variable name found    */
   }
   if  (varinuse < 0)
            errchk(3);                        /* unknown variable name */
   while (inline[col] == ' ') col++;         /* ignore leading blanks */
   if (inline[col] == '=') col++;
       else errchk(8);                        /* missing equal sign */
   parcol = 0;            /* now check for correct parenthesis matchup */
   index = col;
   do {
       if (inline[col] == '(') parcol++;
       if (inline[col++] == ')') parcol--;
       if (parcol < 0) errchk(1);         /* paren count went negative  */
   }
    while (inline[col]);
   if (parcol)
          errchk(2);                 /* left over parenthesis */
   col = index;
```

```
        calcdata(&newval);     /* now go evaluate the full expression */
    if (errcode == 0) {           /* don't update value if error found */
        allvars[varinuse].value = newval;
        disnew(varinuse);                 /* display the changed value  */
    }
}
/* ************************************************************ errout
*/
/* This is the function that displays the blinking error messages on the
*/
/* monitor. Note the extra errors for expansion of the table.
*/
errout()
{
    switch (errcode) {
        case 1 :  errdis("extra right parenthesis ");
                  break;
        case 2 :  errdis("missing right parenthesis");
                  break;
        case 3 :  errdis("unknown variable name ");
                  break;
        case 4 :  errdis("invalid math operator");
                  break;
        case 5 :  errdis("negative value for SQRT");
                  break;
        case 6 :  errdis("Function not found");
                  break;
        case 7 :  errdis("negative value for LOG");
                  break;
        case 8 :  errdis("equal sign missing");
                  break;
        case 9 :  errdis("invalid data field");
                  break;
        case 10 : errdis("division by zero");
                  break;
        case 11 : errdis("File doesn't exist");
                  break;
        case 12 : break;
        case 13 : errdis("Out of memory");
                  break;
        case 14 : errdis("Dash expected");
                  break;
        case 15 : errdis("Invalid format code");
                  break;
        case 16 : errdis("Neg value for FACTORIAL");
                  break;
        case 17 : errdis("Err 17");
                  break;
        default : errdis("unknown error");
                  poscurs(21,70);
                  printf("%d",errcode);
    }
    poscurs(22,12+colerr);
}
/* ************************************************************ initfk()
*/
/* This function is used by APPLIX 1616 to initialise the function keys
*/
/* for the Visual Calculator
*/
char   fkey1[] = { 128, 59, 0 };
char   fkey2[] = { 128, 60, 0 };
char   fkey3[] = { 128, 61, 0 };
char   fkey4[] = { 128, 62, 0 };
char   fkey5[] = { 128, 63, 0 };
char   fkey6[] = { 128, 64, 0 };
```

```
char   fkey7[] = { 128, 65, 0 };
char   fkey8[] = { 128, 66, 0 };
char   fkey9[] = { 128, 67, 0 };
char   fkey10[] = { 128, 68, 0 };

initfk()
{
   DEF_FK( 0, fkey1);
   DEF_FK( 1, fkey2);
   DEF_FK( 2, fkey3);
   DEF_FK( 3, fkey4);
   DEF_FK( 4, fkey5);
   DEF_FK( 5, fkey6);
   DEF_FK( 6, fkey7);
   DEF_FK( 7, fkey8);
   DEF_FK( 8, fkey9);
   DEF_FK( 9, fkey10);
}
/* ****************************************************************/
```

15
The Visual Calculator - Version 1.00

The visual calculator was written to be used for quick calculations of the variety that would ordinarily be done with a hand held calculator. There is no allowance for programming loops, or indirect variables, or any of the other facilities of a modern programming language. There are no complications either, and this program should not require more than a few minutes for the experienced computer user to learn to use, and only slightly longer for the person inexperienced with computers.

It is suggested that you slowly run through the tutorial first, performing the operations suggested, then read the following comments for a description of the visual calculator. This program is intended to be much more comprehensive than the little on-screen calculators that have become popular, but it is not memory resident. Due to the expected future popularity of such programs as "Windows", this program can be as convenient as the present memory resident programs.

15.1 The Visual Calculator Tutorial

1. Copy all files to another working diskette with your operating system or to a single directory on your hard disk.
2. Type VC <return> You will get the beginning screen containing the variable boxes and the help box at the top. The center of the screen contains the transcript box, and at the bottom you will find the Input box.
3. Type A = 123.45 <return> You will find that the value is displayed in the top box and the value will also be displayed at the left of the input equation in the transcript box.
4. Type B = SQRT(A) <return> You will find the square root of A displayed in both places next to the variable B. You may have noticed that the system doesn't care if you use upper or lower case, it forces it to upper case. You now have defined some values for the variables A and B.
5. Type D = 1.23*SIN(SQRT(1.2345 + B*B/A)) <return> Spaces between variables don't matter and you can put them in where you desire to make it look nice. If you get an error message, simply use the left and right cursor keys along with the delete key to fix up the error and hit the return again. You don't even have to be at the end of the line to hit the return.
6. Hit the F6 key then <return> The F6 requests a file to be read in and if you don't specify a filename, it reads in the file named "HELP". This would be a good place to store a list of your other files in the same manner as this file.
7. Hit the F6 key then type AMORT <return> This reads in the file named "AMORT" and calculates each line as it reads it in. Notice that it also changed the names of the variables that it uses to make them more meaningful to you.
8. Type PRINC = 30000 <return> This changes the amount of the loan. We would like to recalculate the payment which we will in the next few steps.
9. Move the arrow up to the line that starts "PMNT=..." by using the up and down arrow keys. When the arrow is point- ing at the line in question,...
10. Hit the F9 key. This moves the line pointed at, by the little arrow, into the input box where it can be modified or used again as is.

11. Hit the <return> key. This will recalculate the payment based on the new principal and the old interest rates and time of repayment. These could also be changed and the payment recalculated.

12. (This was a print function not used by 1616 version).

13. Hit the F6 key again. You will get another prompt for a file name.

14. Type PAYMENT <return> This file will be read in that will give you the results of your mortgage after the first payment. The results will also be printed out.

15. Hit the F6 key again and <return> The last file read in will be reused again and the result of making the second payment will be displayed on the monitor and the printer.

16. Repeat step 15 three or four times.

17. Hit the F1 key. A help screen will appear describing the various math functions available. They can be nested to whatever level you desire.

18. Hit the F2 key. A help screen will appear with a very brief description of the system functions available.

19. Hit the "Home" key. You will be immediately transported to the very top of the transcript where the welcome message was originally seen. The Pgup, Pgdn, Home, and End keys will get you through the transcript window very quickly.

20. Move the little arrow to the line that starts "# A-PRINC", and hit the F4 key once. You will see that the asterisk appears in front of the line. This will "mark" the line. Continuing to hit the F4 key will toggle the asterisk on and off.

21. Move the arrow to the line that starts "# E-EQUITY" and mark this line too.

22. Hit the F5 key The system is now prompting you for a file name to output to.

23. Type STUFF <return> This is simply a filename. Any valid filename could be used. All lines in the transcript box that are "marked" will be output to the file "STUFF".

24. Hit the F6 key and type STUFF <return> All of the lines that were just output will be read in and all calculations will be done.

25. (Turn printing off; not used in 1616 version).

26. Hit the F6 key and <return> The file will be read in again without printing.

27. Hit the F10 key and answer the prompt with Y to end the session.

28. Type VC <return> again to restart the program.

29. Hit the F6 key, type TEST <return> A file with 50 lines will be read in and all calculations performed as an example of the kinds of equations that can be evaluated.

30. Type the following; # I-D J-O K-H L-X <return> This tells the system that we want the variable "I" to print out in Decimal notation, the variable "J" to print out in Octal notation, and "K" and "L" to print out in HeXadecimal notation. (Note - the # must be in the first column.)

31. Type I = 12345 <return> The variable I will be displayed in all three notations in the top box and in decimal notation in the transcript box.

32. Type J = 12345 <return> The variable J will be displayed in Octal notation in the transcript box and on the printer if it is turned on and ready.

33. Type K = 12345 <return> The variable K will be displayed in Hex notation in the transcript box.

34. Type I = 012345 <return> The value of I is read in as an octal value due to the leading zero, but is still displayed as a decimal value.

35. Type J = 0X12345 <return> The value of J is read in as a hexadecimal value due to the leading 0X.

36. Type M = 0XFFFF <return> The variable M is read in as Hexadecimal and displayed in all three formats in the top box, but as decimal in the transcript box. The default display for the integers is decimal.

37. Type I = SQRT(48) <return> The square root is calculated using 15 significant digits and the result is truncated to the next lower value. All calculations are done this way and the result is truncated to the integer value before display.

38. Type A = FACT(170)/FACT(169) - 170 <return> The very small result will indicate to you a measure of the accuracy of calculations. It may not be apparent to you that we are using a factorial function. Calculate the value of FACT(170) to get an idea of the dynamic range available with this system.

39. Hit the F10 key and answer the prompt with Y.

40. Restart the program and try some of your favorite math exercises.

15.2 Additional Comments

1. Files on the distribution disk.

VC.DOC - The file you are reading.

VC.XREL - The executable file for the Visual Calculator (the IBM version is .EXE).

HELP - The users index of files.

AMORT - The loan amortization equations.

PAYMENT - The monthly payment calculations.

TEST - A group of 50 "nonsense" equations.

2. Inputting equations.

All equations are typed into the input box in a normal mathematical expression. Only single valued expressions can be evaluated, no simultaneous equations can be solved with this system.

To raise "A" to the power of "B", use; C = EXP(B*LOG(A)) $ any variables can be used

A dollar sign anywhere in a line renders the remainder of that line as a comment only.

Nesting is allowable to any depth but the entire expres- sion must fit in the input window. Longer expressions must be broken down into smaller statements.

The variables "I" through "J" can be mixed in with the variables "A" through "F" in any manner. The "I" variables are truncated after evaluation so can only be used to store integer values, but that would be acceptable in many cases, such as the original value of the loan in the above example.

3. Naming variables

In order to make the equations easier to read, the names of the variables "A" through "F" can be changed to any names you like with up to 6 characters. The first must be alphabetic and the rest can be alphabetic or numeric. To change the names, use the # sign in the first column of the statement and any order of variable name groups. A variable group is composed of a variable

name "A" through "F", then a minus sign, and finally the new name with no blanks any- where in the group. Any number of blanks can be used between the groups, and you can put as many as you like on one input line, and additional groups on other lines.

Intermixed with the above, or placed on their own input line, you can put as many "base" groups as you like for the variables "I" through "N". A base group consists of the variable name, a minus sign, and one of the letters, "D", "O", "H", or "X".

If, after naming the variables, you wish to rename them to something else, the original names are used for the new name changes. Thus if "A" were named "PLACE" and you wished to rename it to "WHERE" the proper method would be to use "# A-WHERE".

4. Limitations

This version of the Visual Calculator has a limit to the number of lines in the transcript box. There should be enough for most applications. If you need more, I would suggest you change the program.

The limit of numbers is about ten to the plus or minus power of 308. Of course both positive and negative numbers can be used everywhere. The limit for the "I" variables is about 16 million, and can only be zero or positive. The exact number in 2 to the 24th power minus 1. It is the number displayed in the variable "N" when you load the system.

The biggest limitation of the system is the limit of your own creativity. It is up to you to use it in a productive manner or simply to allow it to collect dust like so many of your other programs. I might add that I also have many dust collectors that I have failed to learn to use.

16
Error Messages

17
Coronado Enterprises C Tutor - Ver 1.00

I have no facilities for telephone support of this tutorial and have no plans to institute such. If you find any problems, or if you have any suggestions, please write to me at the address below.

Gordon Dodrill - June 30, 1986

Coronado Enterprises, 12501 Coronado Ave NE, Albuquerque, New Mexico 87122

Chapter 4 - Assignment & Logical Compare
intasign.c Integer assignments (no output)
mortypes.c More data types (no output)
lottypes.c Lots of data types
compares.c Logical compares (no output)
cryptic.c The cryptic constructs (no output)

Chapter 5 - Functions & Scope of variables
sumsqres.c First functions
squares.c Return a value
floatsq.c Floating returns
scope.c Scope of variables
recurson.c Simple Recursion Program
backward.c Another Recursion Program

Chapter 6 - Defines & Macros
define.c Defines
macro.c Macros

Chapter 7 - Strings and Arrays
chrstrg.c Character Strings
strings.c More Character strings
intarray.c Integer Array
bigarray.c Many Arrays (address error)
passback.c Getting data from Functions
multiary.c Multidimensional arrays

Chapter 8 - Pointers
pointer.c Simple Pointers
pointer2.c More pointers
twoway.c Twoway Function Data

Chapter 9 - Standard Input/Output
simpleio.c Simplest standard I/O
singleio.c Single character I/O
betterin.c Better form of single I/O
intin.c Integer input
stringin.c String input
inmem.c In memory I/O conversion
special.c Standard error output

Chapter 10 - File Input/Output
formout.c Formatted output
charout.c Single character output
readchar.c Read single characters
readtext.c Read single words
readgood.c Better read and display
readline.c Read a full line
anyfile.c Read in any file
printdat.c Output to the printer

Chapter 11 - Structures
struct1.c Minimum structure example
struct2.c Array of structures
struct3.c Structures with pointers
nested.c Nested structure (no output)
union1.c An example union
union2.c Another Union example

Chapter 12 - Dynamic Allocation
dynlist.c Simple Dynamic Allocation
bigdynl.c Large Dynamic Allocation
dynlink.c Dynamic Linked List Program (problems)

Chapter 13 - Character and Bit Manipulation
uplow.c Upper/Lower Case Text
charclas.c Character Classification
bitops.c Logical Bit Operations
shifter.c Bit Shifting Operations

Chapter 14 - Example programs
dosex.c DOS call examples (use dosex1616.c instead)
whatnext.c Ask Question in Batch File (no use in 1616)
list.c Source Code Lister
vc.c Visual Calculator

18
HiTech C updates

This is a summary of notes provided by Andrew Morton of Applix Pty Ltd, describing changes made to the HiTech C compiler. Most of the library changes are intended to ensure the output code cooperates with the memory manager.

18.1 Relcc.c

This is the original HiTech `c.c` modified by Colin McCormack and Andrew Morton so that it produces relocatable `.xrel` files straight off, instead of `.exec` fixed position files. The `-r` flag will produce an `.exec` file. This flag needs to be near the front of the `relcc` command line, as there is a bit of an error in the coding of the flag.

The compiler temporary files are placed in the directory `/temp`. You should `assign /temp /rd` or `assign /temp .` or some other safe place before using `relcc`. The compiler has been altered to define the include paths `/hitech` and `/hitech/include` to the prepro- cessor. The identifier `applix1616` is defined to the preprocessor, rather than `applix`, whichcaused problems substituting `#include <applix>`.

`Relcc` fires off the compiler passes using the **exec** system call, rather than searching for them. The compiler passes should reside somewhere in your normal search path (see the `xpath` command in your *Users Manual*.)

`Relcc` has a peek at the Alt C flag, and terminates with an exit code of -1 if an Alt C is detected. You may have to lean on the Alt C to ctach it; the ***exec*** system call clears the flag at the start of each pass.

Each compiler pass closes the standard output, input and error file descriptors upon exit, so they are not available when the next pass is invoked. `Relcc` modifies the ***close*** system call to keep these files open.

The standard directory for include files is `/hitech/include`. Assign this to wherever you actually keep the include files before using the compiler. Specify the flag `-I/hitech/include` to `relcc` (actually the preprocessor) to find everything.

The standard directory for the libraries and the runtime startup code (crtapp, etc) is `/hitech`. Assign this as `assign /hitech /f0/hitech` or wherever the libraries really are.

`Relcc.xrel` is relocatable code, so it hangs about in the top of memory when the compiler passes are crunching. Possibly you will have `make.xrel` somewhere above it also. If the compiler crashes (probably during the assembly pass), you will have to shrink the RAM disk and the stack space to around 100k-150k in total. Sorry about that. Details of how to use `buildmrd` to alter the `mrdrivers` file on your boot disk are given in the *Technical ReferenceManual,* and in the new *User Programs Manual*.

18.2 Modified Files

BUF.C
Altered so that It uses ***malloc()*** instead of ***sbrk()***.

`CLEANUP.C`
Altered so that compiled code supports I/O redirection OK.

CLOSE.C
Now prevents trhe inherited descriptors for standard in, standard out, and standard error from being closed. These are closed by 1616/OS.

GETARGS.C
Altered so that it uses ***malloc()*** instead of ***sbrk()***.

MAKE
An excellent public domain `make` program, which ported very easily. Source code and (microscopic) documentation are in the *Applix Utility Disk #2*.

MALLOC.C
Altered to use the ***getmem*** system call, rather than ***sbrk()***, and all of its own memory management stuff.

MRD_CRTAPP.AS
This is the normal runtime startup code hacked about so that it does not clear the BBS on entry to the code. Use this only for memory resident drivers (MRD) written in C. If the normal startup code is used, the MRD's global storage will get zapped every time the MRD is called by the ***callmrd*** system call. Specify `-jmrd_crtapp.obj` on the command line to fix this. Never use this except when doing an MRD.

NEW_CRTAPP.AS
The runtime code fiddled for normal C programs. It used to leave the environment pointer as a nil pointer. But ***getenv()*** expects it to point to a nil pointer if there is no environment table.

SBRK.AS
The ***sbrk()*** function attempts to allocate more storage beyond the end of the program's BBS segment. Under the 1616's memory system, this will tramp on reserved memory., causing ear-splitting crashes. The modified version here asks the system for the memory before usingit. If the code is an `.xrel` file, then the memory will inevitably be unavailable. If you cannot rework a program to use ***malloc()*** instread of ***sbrk()***, the code will only run as an `.exec` file at $4000, from which it can grow its memory upwards.

SYSCALLS.H
This is a header file which `$define`s every 1616/OS syscall in upper case, with the same usage and argument order as in the *Programmers Manual* Andrew also wrote. Note that the ***printf()***, ***sprintf()***, and ***fprintf()*** calls use a trick which enables them to bypass the normal system call mechanism, so that they may be used with any number of arguments. They work with I/O redirection also. You must use STDERR and STDOUT with ***fprintf()***, not stderr and stdout. Also any file I/O is a bit hairy; do not mix C's file descriptors, reads, writes, etc., with the systems' native ones.

TOUPPER.AS
TOLOWER.AS
Obviously a bad day when these were written. The comparisons were gruesomely wrong. I fixed them up. Watch out for the `#define`'d ***toupper()*** and ***tolower()*** in `ctype.h`. They do not check that the char is in range before performing the addition or subtraction. They are a nuisance.

Index

Table of Contents

Table of Figures