Module 815

Data Structures Using C

Module 815   Data Structures Using C

Aim

After working through this module you should be able to create and use new and complex data types within C programs.

Learning objectives

After working through this module you should be able to:

1. Manipulate character strings in C programs.

2. Declare and manipulate single and multi-dimensional arrays of the C data types.

3. Create, manipulate and manage C pointers to data elements.

4. Create and manage complex data types in C.

5. Use unions to define alternate data sets for use in C programs.

6. Allocate memory to variables dynamically.

7. Manipulate characters and bits.

Content

Strings.

Arrays.

Pointers.

Data definitions – Structures.

Data definitions – Unions.

Dynamic allocation of data

Character and bit manipulation

Learning Strategy

Read the printed module and the assigned readings and complete the exercises as requested.

Assessment

Completion of exercises and the CML test at the end of the module.

References & resources

The C Programming Language. 2nd. edition
Brian W. Kernighan and Dennis M. Ritchie
Prentice-Hall, 1988

Turbo C/C++ Manuals.

Turbo C/C++ MS DOS compiler.

Objective 1   After working through this module you should be able to manipulate character strings in C programs

Strings

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 aesthetically pleasing to you and the people using the output of your program, you need the ability to output text data. We have used strings extensively already, without actually defining them. A complete definition of a string is 'a sequence of char type data terminated by a NULL character,'.

When C is going to use a string of data in some way, either to compare it with another, 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 character (which is usually a character with a zero ASCII code number) is detected. You should also recall (from Module 813: Fundamental Programming Structures in C) that the char type is really a special form of integer – one that stores the ASCII code numbers which represent characters and symbols.

An array (as we shall discover shortly) is a series of homogeneous pieces of data that are all identical in type. The data type can be quite complex as we will see when we get to the section of this module discussing structures. A string is simply a special case of an array, an array of char type data. The best way to see these principles is by use of an example [CHRSTRG.C].

```
#include "stdio.h"
void 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 data declaration for the string appears on line 4. We have used this declaration in previous examples but have not indicated its meaning. The data declaration defines a string called name which has, at most, 5 characters in it. Not only does it define the length of the string, but it also states, through implication, that the characters of the string will be numbered from 0 to 4. 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 have therefore named 5 char type variables: name[0], name[1],

name[2], name[3], and name[4]. You must keep in mind that in C the subscripts actually go from 0 to one less than the number defined in the definition statement. This is a property of the original definition of C and the base limit of the string (or array), i.e. that it always starts at zero, cannot be changed or redefined by the programmer.

Using strings

The variable name is therefore a string which can hold up to 5 characters, but since we need room for the NULL terminating 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 statement which sets NULL equal to zero would allow us to use NULL instead of an actual zero, and this would add greatly to the clarity of the program. It would be very obvious that this was a NULL and not simply a zero for some other purpose.) Now that we have the string, we will simply print it out with some other string data in the output statement.

You will, by now, be familiar with the %s 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; it will then quit. Notice that in the printf statement only the variable 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 found in the next section.)

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].

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.

Some string subroutines

The next example [STRINGS.C] illustrates a few of the more common string handling functions. These functions are found in the C standard library and are defined in the header file *string.h*.

```
#include "stdio.h"
#include "string.h"
void 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)           /* returns 1 if name1 > name2 */
        strcpy(mixed,name1);
    else
        strcpy(mixed,name2);
    printf("The biggest name alpabetically is %s\n" ,mixed);
    strcpy(mixed, name1);
    strcat(mixed,"  ");
    strcat(mixed, name2);
    printf("Both names are %s\n", mixed);
}
```

First, four stringsare defined. Next, a new function that is commonly found in C programs, the *strcpy* function, or string copy function is used. It copies from one string to another until it comes to the NULL character. Remember that the NULL is actually a 0 and is added to the character string by the system. It is easy to remember which one gets copied to which if you think of the function as 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 are 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. The quotes define a *literal string* and are an indication to the compiler that the programmer is defining a string.

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

Alphabetical sorting of strings

The next function to be considered is the *strcmp* or the string compare function. It will return a 1 if the first string is lexicographically larger than the second, zero if they are the two strings are identical, and -1 if the first string is lexicographically smaller than the second. A lexicographical comparison uses the ASCII codes of the characters as the basis for comparison. Therefore, 'A' will be "smaller" than 'Z' because the ASCII code for 'A' is 65 whilst the code for 'Z' is 90. Sometimes however, strange results occur. A string that begins with the letter 'a' is larger than a string that begins with the letter 'Z' since the ASCII code for 'a' is 97 whilst the code for 'Z' is 90.

One of the strings, depending on the result of the compare, is copied into the variable mixed, and therefore 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 ASCII code values of the characters.

Combining strings

The last four statements in the [STRINGS.C] example 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 a single string is produced. 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 which shows both names stored in the one variable called mixed.

Exercise 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.

Write a program thatwill output the characters of a string backwards. For example, given the string "computer", the program will produce

Objective 2   After working through this module you should be able to declare and manipulate single and multi-dimensional arrays of the C data types.

ARRAYS

The last objective discussed the structure of strings which are really special cases of an array. Arrays are a data type that are used to represent a large number of homogeneous values, that is values that are all of the one data type. The data type could be of type char, in which case we have a string. The data type could just as easily be of type int, float or even another array.

An array of integers

The next program [INTARRAY.C] is a short example of using an array of integers.

```
#include "stdio.h"
void 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[index]);
}
```

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. Note carefully that each element of the array is simply an int type variable capable of storing an integer. The only difference between the variables index and values[2], for example, is in the way that you address them. 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.

An array of floating point data

Now for an example of a program [BIGARRAY.C] with an array of float type data. This program also has an extra feature to illustrate how strings can be initialised.

```
#include "stdio.h"
#include "string.h"
char name1[ ] = "First Program Title";
void main( )
{
int index;
int stuff[12];
```

```
float weird[12];
static char name2[] = "Second Program Title";
    for (index = 0; index < 12; index++) {
        stuff[index] = index + 10;
        weird[index] = 12.0 * (index + 7);
    }
    printf("%s\n", name1);
    printf("%s\n\n", name2);
    for (index = 0; index < 12; index++)
        printf("%5d %5d %10.3f\n", index, stuff[index], weird[index]);
}
```

The first line of the program illustrates how to initialise 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 including the terminating NULL. Another string is initialised 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. You can think of a staic declaration as a local constant.

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.

Getting data back from a function

The next program [PASSBACK.C] illustrates how a function can manipulate an array as a variable parameter.

```
#include "stdio.h"
void main( )
{
int index;
int matrix[20];
    /* generate data                                         */
    for (index = 0 ;index < 20; index++)
        matrix[index] = index + 1;
    /* print original data                                   */
    for (index = 0; index < 5 ;index++)
        printf("Start  matrix[%d] = %d\n", index, matrix[index]);
    /* go to a function & modify matrix                      */
    dosome(matrix);
    /* print modified matrix                                 */
    for (index = 0; index < 5 ;index++)
        printf("Back   matrix[%d] = %d\n", index, matrix[index]);
}

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

}

An array of 20 variables named matrix is defined, some nonsense data is assigned to the variables, and the first five values are printed out. Then we call the function *dosome* taking along the entire array as a parameter.

The function dosome has a name in its parentheses also but it uses the local name 'list' for the array. The function needs to be told that it is really getting an array passed to it and that the array is of type int. Line 20 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. In our present illustration, we will use a fixed number of elements to keep it simple.

So far nothing is different from the previous functions we have called except that we have passed more data points to the function this time than we ever have before, having passed 20 integer values. 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 program and 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. Compile and run this program to verify this conclusion.

Arrays pass data both ways

It was stated during our study of functions that when data was passed 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 (that arrays are handled differently from single point data) but it is correct. The reason is that when an array is passed to a function, the address of (or a pointer to) the function is the piece of data that is used. The is analagous to the Pascal construct of placing a VAR in front of a parameter in a procedure definition.

Multiple-dimensional arrays

Arrays need not be one-dimensional as the last three examples have shown, but can have any number of dimensions. Most arrays usually have either one or two dimensions. Higher dimesions have applications

in mathematics but are rarely used in data processing environments. The use of doubly-dimensioned arrays is illustrated in the next program [MULTIARAY.C].

```
#include "stdio.h"
void main( )
{
int i, j;
int big[8][8], large[25][12];
   /* Create big as a a multiplication table              */
   for (i = 0; i < 8; i++)
      for (j = 0; j < 8; j++)
         big[i][j] = i * j;
   /* Create large as an addition table                   */
   for (i = 0; i < 25; i++)
      for (j = 0; j < 12; j++)
         large[i][j] = i + j;

   big[2][6] = large[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 in total. The first element is big[0][0], and the last is big[7][7]. Another array named large is also defined which is not square to illustrate that the array need not be square! Both are filled 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 large 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 demonstrate 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 for which it is being used.

The entire matrix variable big is printed out in a square form so you can check the values to see if they were set the way you expected them to be. You will find many opportunities to use arrays, so do not underestimate the importance of the material in this section.

Exercise 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, for example

         1   2 + 10 = 12
         2   4 + 20 = 24
         3   6 + 30 = 36 etc.

Hint: the print statement will be similar to this:

```
printf("%4d %4d + %4d = %4d\n", index, array1[index],
                          array2[index], arrays[index]);
```

Objective 3   After working through this module you should be able to create manipulate and manage C pointers to data elements.

Pointers

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 examine the next program [POINTER.C] which has some pointers in it.

```
#include "stdio.h"
/* illustration of pointer use                                    */
void main( )
{
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);
}
```

Ignore for the moment the data declaration statement where we define index and two other fields beginning with an 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. 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 a 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!

Two very important rules

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

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 7 as 'pt1 is assigned the value of the address

2. A pointer with a star in front of it refers to the **value** of the variable pointed to by the pointer. Line 10 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 perhaps convenient to think of the asterisk as a star,: it sort of sounds like the word 'store'!

Memory aids

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

Assume for the moment that pt1 and pt2 are pointers (we will see how to define 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 7 of the program assigns the pointer pt1 to point to the variable we have already defined as index because we have assigned the address of 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 10 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 10 therefore assigns to index the value of 13. Anywhere 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.

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 line 8, 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. Line 8 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, anywhere 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.

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 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.

Declaring a pointer

Refer to the fifth line of the program and you will see the 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. Similarly, 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 10 changes the one variable which is displayed three times.

A second pointer program

Consider the next program [POINTER2.C], which illustrates some of the more complex operations that are used in C programs:

```c
#include "stdio.h"
void 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 indentical       */
    two = *(strg+8);
    printf("the second output is %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 variables named *strg* and *list*. We will use them to show the correspondence between pointers and array names.

String variables as pointers

In C a string variable is defined to be simply a pointer to the beginning of a string – this will take some explaining. 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, we can assign the same value to *two* by using the star and the string name. The result of the two assignments are such that *one* now has the same value as *two*, and both contain the character 'T', the first character in the string. Note that it would be incorrect to write 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 9, 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 compiler takes care of indexing for us by automatically adjusting the indexing for the type of variable the pointer is pointing to.(This is why the data type of a variable must be declared before the variable is used.) In this case, the index of 8 is simply added to the pointer value before looking up the desired result because a char type variable is one byte long. If we were using a pointer to an int type variable, the index would be doubled and added to the pointer before looking up the value because an int type variable uses two bytes per value stored. When we get to the section on structures, we will see that a variable can have many, even into the hundreds or thousands, of bytes per variable, but the indexing will be handled automatically for us by the system.

Since *there* is already a pointer, it can be assigned the address of the eleventh element of *strg* by the statement in line 12 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'.

Pointer arithmetic

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

Integer pointers

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 address of the 28th element of the list and print out the same value both ways to illustrate that the system truly will adjust the index for the int type variable. You should spend some time in this program until you feel you fairly well understand these lessons on pointers.

Function data return with a pointer

You may recall that back in the objective dealing with functions it was mentioned that a function could use variable data if the parameter was declared as an array. This works because an array is really a pointer to the array elements. Functons can manipulate variable data if that data is passed to the function as a pointer. The following program [TWOWAY.C] illustrates the general approach used to manipulate variable data in a function.

```
#include "stdio.h"
void fixup(nuts,fruit);   /* prototype the function            */

void main( )
{
int pecans,apples;
   pecans = 100;
   apples = 101;
   printf("The starting values are %d %d\n",pecans,apples);
   fixup(pecans,&apples);
                  /* when we call "fixup" we send the value    */
                  /* of pecans and 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 values are %d %d\n", nuts, *fruit);
   nuts = 135;
   *fruit = 172;
   printf("The values are %d %d\n", nuts ,*fruit);
}
```

There are two variables defined in the main program: *pecans* and *apples*; notice that neither of these is defined as a pointer. We assign values to both of these and print them out, then call the function *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 integer variable and a pointer to an integer variable. This turns out to be very simple. Notice that the parameter 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 heading and the program interface will work just fine.

In the body of the function, we print the two values sent to the function, then modify them and print the new values out. 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 made a copy of the variable pecans 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 for the data. It must be pointed out however, that if you modify the value of the pointer itself in the function, you will have a 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.

Compile and run the program and observe the output.

Programming Exercises

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. Initialise 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.

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

Objective 4   After working through this module you should be able to create and manage complex data types in C.

STRUCTURES

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 collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. Structures are called "records" in some languages, notably Pascal. Structures help organise complicated data, particularly in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities.

The best way to understand a structure is to look at an example [STRUCT1.C].

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

A single compound variable

Let's 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.

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.

Using structure 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.

An array of structures

The next program [STRUCT2.C] 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.

```
#include "stdio.h"
void 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;
    kids[10] = kids[4];                    /* Structure assignment   */
    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);
}
```

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, of course, 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.

Copying structures

C allows you to copy an entire structure with one statement. Line 17 is an example of using a structure assignment. In this statement, all 3 fields of *kids[4]* are copied into their respective fields of *kids[10]*.

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.

Using pointers and structures together

The next program [STRUCT3.C] is an example of using pointers with structures; it is identical to the last program except that it uses pointers for some of the operations.

```
#include "stdio.h"
void main( )
{
struct {
    char initial;
    int age;
```

```
      int grade;
      } kids[12], *point, extra;
   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);
      }
      extra = kids[2];                /* Structure assignment */
      extra = *point;                 /* Structure assignment */
}
```

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.

Pointer arithmetic

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

Now return to the program. 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 stored data to which the pointer points and the construct should be clear. The -> is made up of the minus sign and the greater than sign.

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.

Nested and named structures

The structures we have seen so far have been useful, but very simple. It is possible to define structures containing dozens and even hundreds or thousands of elements but it would be to the programmer's 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 next program [NESTED.C], which shows a *nested* structure.

```
#include "stdio.h"
void 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.descrip.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.

Since we have a new type definition we can use it to define two more variables. The variables *teacher* and *sub* are defined in line 13 to be variables of the type *alldat*, so that each of these two variables contain 5 fields which can store data.

Using fields

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

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.

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.

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. Be conservative at first, and get bolder as you gain experience.

Exercise 4

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 the previous exercise using a pointer to print the data out.

Objective 5   After working through this module you should be able to use unions to define alternate data sets for use in C programs.

Unions

A union is a variable that may hold (at different times) data of different types and sizes. For example, a programmer may wish to define a structure that will record the citation details about a book, and another that records details about a journal. Since a library item can be neither a book or a journal simultaneously, the programmer would declare a library item to be a *union* of book and journal structures. Thus, on one occasion *item* might be used to manipulate book details, and on another occassion, *item* might be used to manipulate journal details. It is up to the programmer, of course, to remember the type of item with which they are dealing.

Examine the next program [UNION1.C] for examples.

```
void main( )
{
union {
    int value;                /* This is the first part of the union
    */
    struct {
       char first;            /* These two values are the second part
    */
       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 computer's 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 is 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 the C compiler 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 also come up in a few of the later examples.

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 next program [UNION2.C] contains 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, in which we define a complete structure, then decide which of the various types can go into it.

```c
#include "stdio.h"
#define AUTO 1
#define BOAT 2
#define PLANE 3
#define SHIP 4
void 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 petrol tank       */
      ford.vehicle_type.car.tires = 5;    /* including the spare    */
      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.vehicle_type.car.tires);
      if (piper_cub.vehicle == AUTO)      /* which it is not       */
         printf("The plane has %d tires.\n",
                  piper_cub.vehicle_type.car.tires);
}
```

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 recognising, but we define no variables at this time.

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 anywhere we would like to. Notice that this does not define any variables, only a new type definition. Capitalising the name is a common preference used by programmers and 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*.

It should be stated 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.

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 frequently, and almost never by novice programmers. You will encounter it occasionally so it is worth your effort to at least know what it is.

Bitfields

The bitfield is a relatively new addition to the C programming language. In the next program [BITFIELD.C] we have a union made up of a single int type variable in line 5 and the structure defined in lines 6 to 10. The structure is composed of three bitfields named *x*, *y*, and *z*. The variable named *x* is only one bit wide, the variable *y* is two bits wide and adjacent to the variable *x*, and the variable *z* is two bits wide and adjacent to *y*. Moreover, because the union causes the bits to be stored in the same memory location as the variable *index*, the variable *x* is the least significant bit of the variable *index*, *y* forms the next two bits, and *z* forms the two high-order bits.

```
#include "stdio.h"
void main( )
{
union {
    int index;
    struct {
        unsigned int x : 1;
        unsigned int y : 2;
        unsigned int z : 2;
    }  bits;
}  number;
    for (number.index = 0; number.index < 20; number.index++) {
        printf("index = %3d, bits = %3d%3d%3d\n", number.index,
                number.bits.z, number.bits.y, number.bits.x);
    }
}
```

Compile and run the program and you will see that as the variable *index* is incremented by 1 each time through the loop, you will see the bitfields of the union counting due to their respective locations within the int definition. One thing must be pointed out: the bitfields must be

defined as parts of an unsigned int or your compiler will issue an error message.

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

Exercise 5

Objective 6   After working through this module you should be able to allocate
memory to variables dynamically.

DYNAMIC ALLOCATION

Dynamic allocation is very intimidating the first time you come across
it, but that need not be. All of the programs up to this point have used
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.) This section discusses how C
uses dynamically allocated variables. They are 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 so that it can
be used by other dynamic variables. As usual, the concept is best
presented by an example [DYNLIST.C].

```c
#include "stdio.h"
#include "stdlib.h"
void 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;
   /* pet2 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 Shepherd");
   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.

Dynamic variable creation

The first program statement is line 11, which assigns something to the pointer *pet1*; it will create a dynamic structure containing three variables. The heart of the statement is the *malloc* function buried in the middle. 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*.

The heap

Every compiler has a set of limitations on it that define how big the executable file can be, how many variables can be used, how long the source file can be, etc. One limitation placed on users by the C compiler is a limit of 64K for the executable code if you happen to be in the small memory model. 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 memory space is limited but still adequate for most programs.

In this model C defines two heaps, the first being called a heap, and the second being called the *far heap*. The heap is an area within the 64K boundary that can store dynamically allocated data and the far heap is an area outside of this 64K boundary 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.

The data and variables are put on the far heap by utilising calls to *farmalloc*, *farcalloc*, etc. and removed through use of the function *farfree*. Study your C compiler's Reference Guide for details of how to use these features.

Segment

C compilers give the user a choice of memory models to use. The user has a choice of using a model with a 64K limitation for either program or data leading to a small fast program or selecting 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 a 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. With C this is only possible by using the tiny memory model. Since a .COM file is already in a memory image format, it can be loaded very quickly whereas a file in an .EXE format must have its addresses relocated as it is loaded. Therefore a tiny memory model can generate a program that 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.

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.

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. The malloc function forms part of the standard library. Its prototype is defined in the *stdlib*.h file, hence this file is included using the statement on line 2. *malloc* 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 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 block of data.

Casting

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 (perhaps most) times 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.

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 come as no surprise to you that these assignment statements look just like assignments 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.

Even though it is not illustrated in this example, you can dynamically allocate and use simple variables such as a single char type variable. This should be used wisely however, since it sometimes is very inefficient. It is only mentioned to point out that there is nothing magic about a data structure that would allow it to be dynamically allocated while simple types could not.

Getting rid of dynamically allocated data

Another new function is used to get rid of the data and free up the space on the heap for reuse. This function is called *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 line 31. 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 DOS, the entire heap will be disposed of with no regard to what we have put on it. The point does need to made that, if you lose a pointer to a block on the heap, it forever removes that block of data storage from our use 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.

Our discussion of the last program has taken a lot of time – 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 few 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 section.

An array of pointers

Our next example [BIGDYNL.C] is very similar to the last, 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*.

```
#include "stdio.h"
#include "stdlib.h"
void 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 print 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]* might seem strange to you so a few words of explanation are 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. 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.

Twelve pointers have now been defined 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, reading 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]* notation. Finally, all 12 blocks of data are release with the *free* function 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.

A linked list

The final example is what some programmers find the most intimidating of all techniques: the dynamically allocated linked list. Examine the next program [DYNLINK.C] to start our examination of lists.

```
#include "stdio.h"
#include "stdlib.h"
#define RECORDS 6
void main( )
{
struct animal {
    char name[25];                  /* The animal's name            */
    char breed[25];                 /* The type of animal           */
    int age;                        /* The animal's 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->age = 4;
    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, "Labrador Retriever");
        point->age = 3;
        prior->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
    */
}
```

The program starts in a similar manner to the previous 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 in line 10, 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.

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.

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 of the list.

The following diagram may help you to understand the structure of the data at this point.

```
start→   struct1
         namef
         breed
         age
         point→   struct2
                   name
                   breed
                   age
                   point→   struct3
                             name
                             breed
                             age
                             point→   …  …   struct7
                                               name
                                               breed
                                               age
                                               point→  NULL
```

It should be clear (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.

Printing the data out

A method similar to the data generation process is used to print the data out. The pointers are initialised 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. Care must be taken to ensure that the last record is not deleted before the NULL is checked; once the data are gone, it is impossible to know if you are finished yet.

It is not difficult, but 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.

Calloc

One more function, the *calloc* function, must be mentioned before closing. 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* in many ways. We will leave you to read about *calloc* as an exercise, and use it if you desire.

Exercise 6

Rewrite STRUCT1.C to dynamically allocate the two structures.

Rewrite STRUCT2.C to dynamically allocate the 12 structures.

Objective 7   After working through this module you should be able to manipulate
characters and bits.

Upper and lower case

The first example program [UPLOW.C] in this section does character
manipulation. More specifically, it changes the case of alphabetic
characters. It illustrates the use of four functions that have to do with
case. Each of these functions is part of the standard library. The
functions are prototyped in the *ctype.h* file, hence its inclusion in line 2
of the program.

```c
#include "stdio.h"
#include "ctype.h"
void mix_up_the_chars(line);
void main( )
{
char line[80];
char *c;
    do {
        /* keep getting lines of text until an empty line is found */
        c = gets(line);             /* get a line of text          */
        if (c != NULL) {
            mix_up_the_chars(line);
        }
    }  while (c != NULL);
}
void mix_up_the_chars(line)
    /* this function turns all upper case characters into lower   */
    /* case, and all lower case to 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);
}
```

It should be no problem for you to study this program on your own and
understand how it works. The four functions on display in this program
are all within the user written function, mix_up_the_chars. Compile and
run the program with data 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.

Many more classification and conversion routines are listed in your C
compiler's Reference Guide.

Classification of characters

We have repeatedly used the backslash n (\n) character for representing a new line. Such indicators are called *escape sequences,* and some of the more commonly used are defined in the following table:

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

A complete list of escape sequences available with your C compiler are listed in your C Reference Guide.

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 next program [CHARCLAS.C] uses the functions that can determine the class of a character, and counts the characters in each class.

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

void count_the_data(line)

void main( )
{
char line[80]
char *c;
    do {
       c = gets(line);              /* get a line of text          */
       if (c != NULL) {
          count_the_data(line);
       }
    } while (c != NULL);
}

void 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, or newline */
         whites++;
      }                                /* end of counting loop    */
   printf("%3d%3d%3d %s", whites, chars, digits, line);
}
```

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?

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 suitable input data.

Logical functions

The functions in this group 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 following example program [BITOPS.C] uses several fields that are combined in each of the ways given above.

```
#include "stdio.h"
void 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 | 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 | 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 data are in hexadecimal format (it is assumed that you already know hexadecimal format if you need to use these operations). Run the program and observe the output.

Shift instructions

The last two operations to be covered in this section are the left shift and the right shift instructions; their use is illustrated in the next example program [SHIFTER.C].

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

The two operations use the following operators:

<< n   Left shift n places.

>> n   Right shift n places.

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

Exercise 7

Using the reference manual of your C compiler, describe any operations on characters and bits that it provides that are not described in this section, particularly those that are described in the *ctype.h* header file.