UNIT 7: POINTERS AND ARRAYS

Structure

7.0 INTRODUCTION

This unit introduces "hard core" C: pointers and their manipulation. Pointers are conceptually quite simple: they're variables that hold the memory addresses of other variables.

To concretise concepts think of an array the elements of which, as you know, are placed in consecutive locations of storage, at regularly increasing addresses. It is apparent that the address of any element is immediately computable relative to that of the zeroth: for, if the address of the zeroth element of the array be x, say, and if each element of it be w bytes wide, then, relatively to its beginning, the address of the kth element must be

x + k * w,k = 1,2. ... , etc.

You can now see that pointers, which are variables to store memory addresses, should be very closely allied to arrays. In an array. The address of any element is one "unit" greater than that of the preceding element. Different types of "units" - chars, ints, floats or doubles, or user-defined types - would of course have differing numbers of bytes inside them. So going from one array element to the next (by incrementing its subscript) is equivalent to "incrementing" the pointer to the current element; incrementation here meaning increasing the value of the pointer by the width, in bytes, of the object it was pointing at. SO pointers can provide an elegant, and often faster, alternative to array manipulation.

At this point there is one question that might occur to you: why are pointer variables necessary at all? Many languages such as FORTRAN and COBOL get by quite happily without them. And it is at pointers that the correspondence between C and Pascal begins to fade away. Though Pascal does have pointers, it is only C that has provided them with such power, elegance, flexibility and case of manipulation, as to set it apart from all other programming languages.

Quite a part from providing a faster alternative to array manipulation, pointers are used in C to dynamically allocate memory, i.e. while a program is executing. This aspect, as we have hinted before, is important in situations where it is impossible to know in advance the amount of storage required to store the data to be generated, or which will be input. In Program 6.14 we did not know in advance how much space to set aside for primes upto hundred million of the type k * k + 1. In a sense we gambled: we chose, without any evidence, the array dimension to be 1000. Declaring too large an array may waste memory; but too small an array might cause the program to run haywire or be aborted. Dynamical allocation gets around this problem by enabling as much memory to be created as required, when required. If memory can't be found, the allocating function can inform the program accordingly.

The third use to which C puts pointers is as arguments to functions. If a pointer to a variable is passed as an argument to a function, in effect the function has been informed where in memory the variable is located.

The function can then alter its value. Functions can manipulate and change the contents of their pointer arguments: in the next Unit we will see that these are in fact the only arguments that functions can modify.

Finally, C uses pointers to represent and manipulate complex data structures.

scarf ( ) is an extremely versatile function that C uses for keyboard input. In this unit we will explore some of its further properties. And finally we shall look at multi-dimensional arrays where more than one subscript is required to locate an element; matrices and magic squares are common examples of two dimensional arrays.

7.1 OBJECTIVES

After reading this unit you should be able to

- understand and use pointers
- see the underlying unity of pointers and arrays
- use the subscript operator
- exploit the versatility of scanf ()
- use the string I/0 functions puts () and gets ()
- handle multidimensional arrays

7.2 POINTER VARIABLES AND POINTER ARITHMETIC

One of the most powerful features of C, and one that makes it quite close to assembly language. is its ability to refer to the addresses of program variables. In C one can in fact declare variables to be pointers, that is variables which will hold the memory addresses of other variables.

The declaration

char *x;

declares x to be a pointer; x can now be assigned die address of a char-like variable. Similarly one can declare pointer variables to point to (i.e. hold the addresses of) ints, or longs, or floats or doubles, in short, of variables of any type, including user-defined types.

The statements

```
int *x, *y, z = 10;
long *p,q;
float *s;
double *t;
```

declare

i) x and y to be pointer variables of type int. That is to say, x and y can hold the addresses of int like variables, such as z;

ii) p is a pointer to longs. It can store the address of a long int, for instance of q;

iii) s can store the addresses of float variables;

iv) t is a pointer to double.

The values of pointer variables are memory addresses.

How does one extract the memory address of a program variable? And, conversely, given a memory address, how does one determine the contents at that address? To answer these questions, let's first recall our picture of memory: a sequence of boxes, each identified by a unique positive number or address, with the addresses written to the left of the boxes. A C variable has precisely these two parts associated with it. its value or address, fixed once and for all after compilation and linkage, and its value, the contents of the box, which may vary as the program executes. Given a variable, C allows its address to be extracted; conversely, given an address, it is possible to obtain the contents at that address.

Let's 1ook again in at the first of the declarations above:

```
int *x, *y, z = 10;
```

The C "address of" operator is the ampersand, &. It's a unary operator, that, applied to a variable (strictly, to an value), yields its memory address:

address of z == &z

&z is the address at which the computer stores the int variable z, and this remains fixed throughout program execution. Attempting to change it in any way is an error.

This address may be assigned to x, because x has been declared to be a pointer to variables of type int:

```
x = &z;
```

x now holds the address of z, which you may actually print:

```
print f ("The address of z is: \t%u\n", x);
```

As remarked above, the "address or" operator can only be applied to values. Like all unary operators it has a priority just below the parentheses operator, and groups from right to left. The "address of" operator is not unfamiliar: the scanf () function has an argument list of pointers to memory locations.

Reciprocally C has a "contents at address", or "dereferencing" operator, the asterisk, *. This unary operator yields the value resident at a memory address. Applied to a pointer variable which has been initialized to an address, it finds the contents at that address.

In die example above, the assignment:

x = &z;

makes x holds the address of int z. *x gives the contents at the address that's held in x; the output from the printf () below will therefore be the value of z, 10:

printf ("The contents of the address held in x are:\t%dn",*x); See Program 7.1.

```
/* Program 7.1 */
#include
main ( )
        {
         int * x, z = 10;
         x = &z;
         printf ("The address of z is.\t%u\n", x);
         priiitf ("The contents of the address held in x are:\t%d\n", *x);
         *x = 20.
         printf ("The new value of z is:\t%d\n", z);
        }
```

To clarify these concepts further, let's assume that the address of the int variable z is 38790, and that the contents at this address are 10. See Fig. I below. In the assignment:

x = &z;

x gets the value "address of z", i.e. 38790. So, if the address of x itself is 56780, the contents at 56780 will be 38790.

Address	Contents of address
Z, located at 38790	10
Other prorgram variables " "	
X, lacated at 56780	38790

The contents at x is the address of z
The contents of z is the value 10

Figure II

The subsequent assignment:

* x= 20

Changes the value of the int object whose address is held in x, namely z, to 20. Here's the picture.

Address	Contents of address
Z, located at 38790	20
Other prorgram variables " "	
X, lacated at 56780	38790

Figure II - The value of z modifed indirectly

Let's now look at Program 7.2. c1, c2, c3, ..., c9 are char variables that have been assigned the initial values 'A', 'b', 'h', etc., pointer is a pointer to char. The first printf () outputs the current values of c1, c2, ..., c9. Then pointer is assigned the address of c2:

Pointer = & c2;

In the next assignment the contents currently resident at pointer are altered to 'p':

* pointer= 'p';

c2 therefore now has the value 'p'. One could of course replace the two assignments:

pointer = & c2, * pointer = 'p';

by the single assignment:

c2 = 'P';

but that doesn't show the pointers at the back of it. Pointers do the same. albeit indirectly manipulating values, so to speak, from a distance. the way a puppeteer dances her puppets. In Program 7.2 pointer is made to point successively at c2, c3,..., c9 and to make new assignments to them.

```
/* Program 7.2 */
#include
main ( )
        {
          char c1 = 'A', c2 = 'b', c3 = 'h'. c4 = 'i'
          c5 = 's', c6 = 'h', c7 = 'e', c8 = 'k', c9 = ' ';
          char *Pointer;
          printf ("%c%c%C%C%C%C%C%C%C", c1, c2, c3, c4,
         c5, c6, c7, c8, c9);
          pointer = &c2;
          /* contd. */

           *pointer = 'p';
          pointer    = &c3;
          *Pointer  = 'a';
          pointer    = &c4;
          *Pointer  = 'r';
          pointer    = &c5;
          *Pointer  = 'a';
          pointer    = &c6;
          *pointer  = 'j';
          pointer    = &c7;
          *Pointer  = 'i';
          pointer    = &c8;
          *Pointer 't';
          pointer    = &c9;
          *Pointer  = 'a';
         printf ('%c%c%c%c%c%c%c%c%c", cl, c2, c3, c4, c5, c6,
        c7, c8. c9);
         printf ("\n");
        }
```

In Program 7.3 the pointers to int p, q and r are assigned the addresses of the ints x, y end z, respectively. Then the contents of the addresses in p and q are set to 5 and 3, which become the respective values of x and y. Post- incrementation of x and post- decrementation of y change them to 6 and 2, these are the new values of * p and * q. Therefore the expression:

***r= *p* *q + *p/*q**

makes the contents at the address held in r to be 15. This is the valuethat z gets.

```
/* Program 7.3 */
include
main ( )
        {
          int x, y, z;
          int *p, *q, *r;
          p = &X;
          q = &Y;
          r = &z;
          *p= 5.;
          *q= 3;
          x++;
          y--;
          *r= *p * *q + *p *q;
          printf ("The contents at the address in p are %d\n', *p);
          printf ("The contents at the address in q are %d\n', *q);
          printf ("The contents at the address in r are %d\n', *r);
        }
```

If you think for a moment about how pointer variables are declared, with separate declarations for each type, a question that may occur to you is the following: addresses, whether they be of char, int, float or double variables, must each be very like one another. An address is an address is an address. Can the address of a char be qualitatively different from that of an int, or of a double from that of a long? Then why must pointer declarations distinguish between addresses 'of
variables of differing data types? In other words, why couldn't the inventor of C do with just one declaration for pointer variables, such as for example:

```
pntr c, x, y, p, s, t;
```

rather than separate declarations for each type:

```
char *c;
int *x, *y, z = 10;
long *p, q;
float *s;
double *t;
```

The fact is that there is a subtle difference between pointers which refer to variables of different types. By informing the compiler about the type of object being referenced by the pointer, the pointer can be "scaled" appropriately to point to the next object of the same type. Recall that in the IBM PC for example, a char value sits in a single byte of memory, an int in two bytes, a float in four and a double in eight. So when a char pointer is "dereferenced", it should give the contents of the single byte whose address it holds. But when an int pointer is dereferenced, it must give the contents of two consecutive bytes (regarded as an integral unit, a word), because an int is two bytes big. Similarly, when a pointer to float is dereferenced, it must yield the contents of the set of four bytes (two words) which hold that float variable. Thus, by associating a type with a pointer variable it helps to dereference the value of the referenced variable properly in accordance with its size.

There is another reason why it makes sense to distinguish between pointers to different data types. We have seen that there is a close relationship between pointers and arrays in C. Stepping from one array element to the next directly translates to incrementing a pointer which holds the address of the first element of the array, as shown in Figure II below:

Incrementation changes pointer by width of element it points at.
Incrementing a pointer to char increases it by 1.
Incrementing a pointer to double increases it by 8.

Figure III

Now, a string array is a collection of individual bytes. The address of any one byte is one greater than the address of the preceding, So in going from one character of the string to the next, you increment the pointer to that byte. But if you have an array of doubles, then, in going from one element of the array to the next, again incrementing the pointer that stores the address of that element, you would actually increase the pointer's value by eight, because doubles are eight bytes wide! Adding one to a pointer produces different results if the pointer is a char or int or float or of any other type. So pointers are not all alike. Neither are they integers in the sense of int. Incrementing an int increases its value by one. Incrementing a pointer increases its value by the width of the object it points to. For an IBM PC incrementing a pointer to int increases its value by 2! See Fig. II., and the output of Program 7.4, which you should study carefully. Because pointers are not ints, C does not allow many of the operations on pointers, that are possible with ints: for example, you can't multiply two pointers together; nor should you want to: pointers are addresses, much like the house numbers on a street. What could one possibly want to multiply house numbers for? The only valid operations on pointers are:

i) assignment of pointers of similar type, using a cast operator if need be;

ii) adding an integer to, or subtracting an integer from, a pointer;

iii) subtracting or comparing two pointers that point to elements of the same array; and

iv) assigning or comparing a pointer to null.

A null pointer points nowhere.

In Program 7.4 four pointers are assigned the respective addresses of four variables of the basic types. The output shows that adding 1 to each pointer increased it by the width of the type it pointed at.

```
/* Program 7.4 */
# include
main ( )
{
        char *char-Pointer, char-Variable;
        /* contd. */
        Int * int_pointer, int-variable;
        float*float_pointer, float_variable;
        double *double_pointer, double-variable;
        char_pointer = &char variable;
        int-pointer = &int-variable;
        float-pointer = &float-variable;
        double-pointer = &double-variable;
        prinitf ("Address of char-variable: %u, Address + 1: %u\n",
                        char -pointer, char_pointer + 1);
        printf ("Address of int-variable: %u, Address + 1: %u\n",
                        int-pointer, int_pointer+ 1);
        printf ("Address of float-variable: %u, Address + 1: %u\n",
                        float-Pointer, float-pointer + 1);
```

```
    printf ("Address of double-variable: %u, Address + 1: %u\n",
                        double_pointer, double-pointer + 1);
    printf ("\nMoral: Adding 1 to a pointer does not  necessarily\n");
    printf ("increase it by 1. Pointers are not ints.\n");
}
```

Here is the output of Program 7.4 above on my computer. Very probably it will be different on yours.

```
/* Program 7.4. Output */

Address of char-variable: 65523, Address + 1: 65524
Address of int-variable: 65518, Address + 1: 65520
Address of float-variable: 65514, Address + 1: 65518
Address of double-variable: 65506, Address + 1: 65514

Moral: Adding 1 to a pointer does not necessarily
increase it by 1. Pointers are not ints.
```

Check Your Progress 1

Question 1: Create, compile and execute Programs 7.1, 7.2, 7.3 and 7.4.

Question 2: Pointer hold the addresses of program variables. In turn, they have addresses too. Modify Program 7.1 to determine the address at which the pointer x is stored.

Question 3: Suppose x is a pointer to int, initialized to point to an int y, which has been assigned the value 11. Which of the following expressions are equivalent? Which do not change y?

x++,(x++), *++x,*(++x)

7.3 POINTERS, ARRAYS AND THE SUBSCRIPT OPERATOR

We have already remarked upon the close relationship between Pointers and arrays. That relationship is crystallised in the following truth:

The name of an array is the pointer to its zeroth element.

Let primes [1000] be an array of ints. Then

primes == & primes [0]

The name primes is itself a pointer! Its value is the address of primes [0]. primes + 1 is the address of the next element of the array, primes + i is the address of its (i + 1)the element. One can use this property in the following way: suppose you wish to scan values into the elements of for (i = 0; i < 40; i ++)

```
scanf ("%d",(marks + i));
```

This statement works because the arguments of scanf () must be pointers, and marks + i is, pointer to the ith element after the zeroth of marks. Compare this statement with the corresponding one in Program 6.13:

```
for (i = 0; i < 40; i ++)
    scanf ("%d", & marks [i])
```

Program 7.5 uses the pointer notation to store the first 1000 primes in an array, int primes [1000].

```
/* Program 7.5 */
#define TRUE 1
#define FALSE 0
#include
main ( )
        {
        int primes [1000] number;
        int i = 0. factor, isprime;
        * primes = 2;
        for (number=3;;nuinber+=2)
            {
            isprime = TRUE;
            for ((factor = 3., factor * factor <= number; factor += 2)
            if (number % factor == 0)
                isprime = FALSE;
            if (isprime)
                 {
                * (primes + ++ i) = number;
                if (i == 1000)
                   break;
                 }
            }
        }
```

The declaration:

```
nt princes [1000];
```

creates spice for 1000 primes beginning at the memory address primes, primes is a pointer, as is (primes + ++i). But the alternative declaration:

```
int * primes;
```

in this program wouldn't have worked; because it merely defines a pointer to int; it does not create storage to pile the 1000 primes in, as they are generated. If you do attempt to store them heedlessly, using a statement like:

```
* (primes + ++ i) = number;
```

you would be writing them in places where the compiler does not expect them. In turn, it will give you unexpected results!

The commonest types of char arrays in C are strings. Strings are very convenient to manipulate for an especial reason: A C string is a pointer to itself! In plain words, the string:

```
"I am a pointer."
```

is a pointer it points to the byte which contains the first character of the string, namely the letter 'I'. As in all strings, the last byte here is of course the null character. To prove that this spring is in fact a pointer pointing to the byte containing 'I', execute the following simple program:

```
/* Program 7.6 */
    # include
    main ( )
```

```
{
 int i = 0;
 for (; * ("I am a pointer." + i)!='\0';i++)
      putchar (* ("I am a pointer." + i));
}
```

The operation..

```
* ("I am a pointer." + i)
```

extracts the ith character of the string. For some value of i this will become the null character. At that time the Boolean of the for (;;) loop will have become false, and the loop will be exited.

Precisely because a string is a pointer to itself, and is therefore a memory address, it is possible to print the addresses of the bytes in which the string is stored. Program 7.7 does just this. We use the %u format conversion character, since memory addresses are unsigned quantities.

```
/* Program 7.7 */
# include
main ( )
{
 char * fact = "We are the best.";
 for (;*fact!=i\O';fact++)
      printf ("%c %u\n",* fact, fact);
}
```

Given the fact that a string is a pointer to itself, the definition:

```
char* fact = "We are still the best.";
```

makes sense. fact now points to the first byte of the string. But a subsequent assignment to fact:

```
fact = "We will always be the best.";
```

will cause it to point to a different address of memory, the address where this new string begins. It is important for you to realize that the declaration of fact allocates enough memory to hold not merely the pointer variable fact but also the entire length of the string. The output of Program 7.8 should convince you of this truth. It prints the address of fact, and the addresses of each byte of the string to which it points.

```
/* Program 7.8 */
#include
main ( )
{
 char * fact = "We will always be the best.";
 printf ("Bytes allocated for fact: %d\n", sizeof fact);
 printf ("Memory address of fact: %u\n", & fact);
 for (printf("String contents: \n");*fact!='\0';fact++)
      printf ("\t\t%c is at %u\n". * fact, fact);
}
```

Contrast this with a string array, defined in classic C thus:

```
static char siring-array [] "An array of chars";
```

Here the address string_array is fixed when the program has been compiled and linked. The size of string_array [] is just big enough to hold each byte in the string, including its invisible terminator. This

means that though each element of string_array [] may be altered, the pointer string-array cannot be assigned any other value. So the incrementation:

```
String -array ++;/*        WRONG,
                           because string-array is a fixed address */
```

is taboo; compare this with:

```
fact ++;/*                 ACCEPTABLE,
                           because fact is a  pointer*/
```

in Programs 7.5 and 7.6, in which fact was declared as a pointer to a string; but it could be assigned the address of any char. String-array is a pointer to the array of that name, sitting at a fixed address.

To recapitulate; the definition:

```
char * x = "We will always be the best."
```

defines a pointer variable x, and allocates memory to store the string, as well as the pointer to it. x is a variable. It may be given other values as the program executes, die address of any other char. On the other hand, the definition:

```
static char x[] = "We will always be the best";
```

defines an array at a fixed memory location. X is a constant pointer: it points to the zeroth element of x []. A new assignment cannot be made to x. Inasmuch as pointers can he used interchangeably with arrays (because, for any array a [],a + i == & a [i]), the converse is also true. For any pointer p, the quantity p [i] has, a meaning. It is the value of the ith object from the one that p points to, of the same type. Regarded in this way the symbols [] constitute an operator called the subscript operator. This operator associates from left to right and has a priority as high as the parentheses operator. Now it is not quite necessary that the ith object from the one referenced by p be "ahead of"or "before it" in memory: in other words, the subscript i may be a positive or negative number! It is in this sense only that C allows negative subscripts. Program 7.9 has an example of the subscript operator.

```
/* Program 7.9 */
# include
main ( )
        {
          int i;
          char * fact = "We are the best.:
          for (i=O;fact[i]!='\0;i++)
                putchar (fact [i]);
        }

              /* Program 7.9: Output */
                 We are the best.
```

Given that strings are pointers, it must now be clear that wherever printf () is deployed to print a string, its string argument may be replaced by a pointer to the string. Conversely because strings are arrays, whenever scanf () reads a string, a character array must be in place to hold the characters entered. Consider Program 7.10:

```
/ * Program 7.10 */
# include
```

```
main ( )
        {
          char tree-1 [20], tree-2 [20], such-3 [20], tree-4 [20];
          printf ("Name your favourite tree:");
          scanf ("%s", tree-1);
          printf ("Name another tree that you like");
          scanf ("%s", tree % 2);
          printf ("Name a third:");
          scaiif ("%s", tree-3);
          printf ("Name a fourth tree that you appreciate:");
          scatif ("%s", tree-4);
          /* contd. */


           printf ("These are your favourite trees: %s, %s, %s and %.%\n",
                                        tree-1, tree-2, tree-3, tree-4);
          printf ("You\'re a good person!!! \n");
        }
```

Check Your Progress 2

Question 1: Create, compile and execute Programs 7.5 - 7.10.

Question 2: Give the output of the following programs:

```
/* Program 7.11*/
#include
main ( )
        {
         int i,*j;
         static int primes [] = {
                     2,3,5,7,11
                   }
         j = primes;
         for (i = 0; i < size of primes / size of (int); i ++)
           printf ( "%d/n", *j++),
         for (j=&-primes [4]; j = & primes [O];)
           printf ("%d/n", --* j --);
        }

/* Program 7.12 */
#include
main ( )
        {
         int I, * j;
         static int primes [] = {
                     2, 3, 5, 7, 11
                   };
          j = primes + 4;
         for (; j >= primes + 1; j --)
           printf ("%d/n", j[0]);
         for (j = primes, i=0; i<4;)
           printf ("%d/n", j [++i]);
         j = primes +4;
```

```
    for (I = 0; I< = 4; I++)
        printf ("%d/n", j [-I]);
}
```

7.4 A DIGRESSION ON scanf ()

When scanf () (with the %s format conversion character) examines the strewn of characters arriving from the input, it ignores all leading white space characters _ blanks, tabs, new lines or form feeds, so in Program 7.10 if you responded with:

< SPACE > < SPACE > < SPACE > < CR > < SPACE >Banyan <CR >

the array tree_1 [] would still contain the seven characters 'B', 'a', 'n', 'y', 'a', 'n' and '\0' only. When a string of characters is read in via %s, then not only does scanf () ignore any initial white space characters. It also stops reading further as soon as a white space character is encountered; this you may verify by entering:

Ficus Religiosa

for the name of your favourite tree in Program 7.10. The word "Ficus", you will note, is stored in tree_1 [], "Religiosa" in tree_[2]! But if a number is sandwiched between the % and the s, for example %6s, then the specifics number of characters is read, unless a white space character is encountered first. Executing Program 7.10 teaches us that leading white space characters in the input to scanf () with %s are ignored, while embedded white spice characters cause it to stop reading further input; but there are two noteworthy cases, which we'll examine in turn. The first arises when %c is used as die format conversion specifier. Then every single character - including any white space characters - in the input is read. Consider:

scanf ("%d%c%d", &x, &y, &z);

and suppose the input to this statement was:

5m6

then x would get the value 5, y the value 'm' and z the value 6 respectively. But if instead the input was:

< SPACE >7< SPACE >m< SPACE >8

the value assigned to x would be 7, y would be assigned the char value " (SPACE), and z would retain its former value of 6. The reason for this behaviour is as follows: in reading an int with %d 'scanf () ignores all leading white space characters; when it receives the input 7 it assigns it to x. With the %c format conversion character, each keystroke is significant. If < SPACE > is typed next, that is the value that y gets. Then m is received, but the corresponding format specifier (%d) expects an int value. Now scanf (stops reading as soon as it encounters a character that is not valid for the type being read. The attempt to assign the value m to z fails. This is illustrated in Program 7.13, appended below, and a sample output:

```
/* Program 7.13*/
#include
main ( )
        {
          int x, z;
          char y;
          printf ("Type in the values 5m6 for x, y and z, without spaces:")
          scanf ("%d%c%d", &x, &y, &z);
          printf ("x = %d, y = %c, z = %d\n", x, y, z);
          printf ("Type in the values 7 < SPACE >M< SPACE >8 for x, y and z:
          scanf ("%d%c%d", &x, &y, &z.);
```

```
    printf ("Now x = %d, y = %c, z = %d\n", x, y, z);
}
```

```
        /* Program 7.13: Output */
```

Type in the values 5m6 for x, y and z, without spaces: 5in6 .
x = 5, y = m, z = 6

Type in the values 7< SPACE >m< SPACE >8 for x, y and z: 7 m 8

Nowx=7,y= z=6

Programs 7.9 and 7.10 would have behaved somewhat differently if the character had been included inside the control string, separating %c from the second %d:

```
        scanf ("%d%c %d", &x, &y, &z);
```

Then an arbitrary number of blanks are permitted in the input to separate the value entered for y from that for z. This is illustrated by the two calls to scanf (in Program 7.14. Note the difference in their control strings. Then note the difference in the assignments they made (or could not make) to x, y and z.

```
/* Program 7.14 */
#include
main ( )
    {
      int x, z;
    /* contd. */

    char y;
    printf ("Enter values for x, y and z, using spaces for separators\n");
    printf ("Remember to enter a non-white space character for y: ");
    scanf ("%d %c %d", &,x, &y, &z);
    printf ("x = %d, y = %c, z = %d", x, y, z);
    printf ("\n Enter values for x, y and z, but this time \n");
    printf ("no spaces between values for x and y: ");
    scanf ("%d %c %d", &x, &y, &z),
    printf ("x = %d, y = %c, z = %d", x, y, z);
}
```

```
        /* Program 7.14. Output */
```

Enter values for x, y and z, using spaces for separators Remember to enter a non-white spice character for y: 34 b 45
x = 34, y = b, z = 45 Enter values for x, y and z, but this time no spaces between values for x and y: 45 b 56
/* We typed spaces! */ x = 45, y = , z = 45

The last line of the output above shows that scanf () terminates reading the input stream as soon as it encounter therein a character invalid for the type listed (it was given a 'h' when it expected a numeric value for z, and did not read the 56 meant for it). For decimal ints valid characters are an optional sign, followed by the digits 0 - 9; for octals the digits 0 - 7; and for hex the digits 0 - 9, a - f or A - F. A field width (e.g. %4d* %6c, %3o, %2x, etc.) may be specified: reading stops as soon as the specified number of characters has been read in, or, as we noted in the last example, a character is entered that does not match the type being read. When a field width is supplied with %c, it is assumed that the corresponding argument is a pointer to a character array, at least as big as the field width specified. See Program 7.20.

The second case in which white spaces in the input to a scanf () can be made significant is through a bracketed string read, explained below.

We learnt from Program 7.10 that the %s specification reads a sequence of characters of which the first must be a non-white space character; and reading terminates as soon as a white spice character is met with. The corresponding argument must be a character array, at least one byte bigger than the sequence of non-white space characters entered, to hold the termination null character: scanf() automatically appends it when it's done reading. If a field width is specified, then no more characters than that width can be read in. Given this propensity of scanf () to stop reading a string as soon as it encounters a white space character inside it, how can a string such as:

"I am a string."

be stored via scanf ()? For this purpose one uses a bracketed string read, %[..] where the square brackets [] are used to enclose all characters which are permissible in the input. If any character other than those listed within the brackets occurs in the input string, further reading is terminated. Reciprocally, one may specify within the brackets those characters which, if found in the input, will cause further reading of the string to be terminated. Such input terminators must be preceded by the caret character (^). For example, if the tilde symbol (~) is used to end a string, the following scanf() will do:

```
char string [80];
scanf ("%[^~]", string);
```

Then, if the input for string consists of embedded spaces; no matter: they will all be accepted by scanf (); and reading will stop when a tilde (~) is entered. This is illustrated in Program 7.15 and its output:

```
/* Program 7.15 */
#include
main ( )
        {
          char string [80];
        /* contd. */

           printf ("Enter a string, terminate with a tilde (~).....");
           scan f (" % [^~] ", string);
           printf ("%s", string);
        }

                /* Program 7.15: Output */

Enter a string, terminate with a tilde (~) ... I am a string. ~
I am a string.
```

Though the terminating tilde is not itself included as an element of the string read. it stays in the "read buffer"__ the area of memory designated to store the input - and will be picked up by the next call to scanf (), even though you may not want it! This is illustrated by Program 7.16 and its output. There, the second call to scanf (is executed utomatically, and the "dangling" tilde is assigned to die char x. The call to putchar ( ) prints the value of x:

```
/* Pr6gram 7.16 */
#include
main ( )
        {
          char string [80];
          char x;
          printf ("Enter a string, terminate with a tilde (~)...");
          scanf ("%[^~]", string);
          scanf ("%c", &x); /* The leftover from the last scanf
                  is read here. This scanf ( ) doesn't
```

```
            wait for you to enter another char.
    printf ("%s", string);
    putchar (x);
}

        /* Program 7.16: Output */
```

Enter a string, terminate with a tilde (~) ... 1 am a string. ~
I am a siring. ~

Compile and execute Program 7.16. You will find that the machine executes the second scant ( ) without so much as a by-your-leave! Such dangling characters must be "absorbed away" by a subsequent call to scanf ( ) with %c, or to get char ( ) else they may interfere in unexpected ways with subsequent calls to scanf ( ) or get char ( ).

If there are non-format characters within the control string of a scanf ( ), they must be matched in the input. For example, consider the scanf (0) we'd used in Program 5.16:

```
scanf ("%d-%d-%d", &Day, &Month, &Year);
```

The hyphens in the control string are non-format characters. Their presence implied that three ints were to be read in, with their values to be separated by a single intervening hyphen. Using any other separators would have led to error. To execute Program 7.17 below correctly, you would have to admit that you like C!

```
/* Program 7.17 */
#include
main ( )
    {
      int x. y;
      printf ("Type \"I like C!\", then values for x and y...");
      if (scanf ("I like C! %d %d", &x, &y) == 2)
        printf ("Thank you! The sum of x and y is: %d\n", x + y);
      else
      printf ("I will not add those numbers for you!\n");
    }
```

Program 7.17 uses another interesting property of scanf (, one we've seen before (refer to Program 5.16): that it returns the number of items that it has successfully read. If this does

not equal the number expected by the program, an error message can be output.

An asterisk (*) in the control string of a scanf (), placed between the % sign and the format conversion character which follows it. causes the corresponding input to be ignored. Consider the scanf ():

```
scanf ("%f %*f %f", &x, &z); Suppose the input was:

        12.34 45.67 78.90
```

Then x and z would get the values 12.34 and 78.90 respectively.

The valid characters for an input of floats are an optionally signed sequence of decimal digits, followed by an optional decimal point and another sequence of decimal digits, followed if need be by the letter e or E and an exponent, which may be signed. The modifier h with the d or f specifiers is used to read in and assign longs and doubles respectively. The modifier h with the d specifiers reads short ints. These, rules are summarized below:

%d reads ints in decimal notation: argument must be a pointer to int;
%Id reads long ints: argument must be a pointer to long;
%hd reads short into: argument must be a pointer to short;
%u reads unsigned ints: argument must be a pointer to unsigned;
%o reads numbers in octal notation,

%lo reads long octals;
%ho reads short octals;
%x reads hex ints;
%lx reads long hex ints;
%hx reads short hex ints;
%e,%f and %g read numbers expressed in floating point notation;
%le, %if, %lg require arguments to point to double;
%c reads single characters; argument is a pointer to char;

Akin to putchar () and getchar (), C provides in addition to printf () and scanf () other functions for greater convenience of string I/O. These are puts () and gets ().

puts () outputs its string argument on the terminal's screen:

puts ("This is an easy function to use, isn\'t it?");

Again, the argument of puts () may be the name of a string array. Consider the following program:

```
/* Program 7.18 */
#include
main ( )
        {
          static char string-1 []= "I hope that you enjoy C.";
          static char string_2 []= "It\s great fun!"
          puts (string_1);
          puts (string_2);
        }
```

Neither of the arrays string_1 [] or string_2 [] ends with a newline; yet the output of the program is printed in two lines, with the cursor stationed on the third, as you can see by executing the program. The puts () function replaces the terminating null character of its string argument by a newline.

gets () gets the characters that you type in at the keyboard, until is pressed. This bunch of characters is stored as a string array, for which memory must be allocated before gets () is invoked. In Program 7.19 below that array of 55 cliars is get_string []. The indicating end of keyboard input is not stored in the array; gets () replaces it by a null character, marking the end of the input string.

```
/* Program 7.19 */
#include
main ( )
        {
          char get-String [55];
          puts ("Tell me how you are");
          puts ("in 55 keystrokes or less.");
          puts ("That \'s inclusive of the ,")
          puts ("which we expect you to type presently!");
          printf ("%s", "Begin here:");
          gets (get-String);
          puts ("Thanks for that detailed description.");
```

```
}
```

gets () does more than merely getting a string from the keyboard. If it was able to read the input string successfully, it returns the address of the array to which the string is assigned; if not. or if no characters are read, it returns the null pointer, which by convention has the value 0, and points nowhere.

Check Your Progress 3

Question 1: Describe what happens if you input more than 55 characters to program 7.19

Question 2: Give the output of the following program, in which a field width is specified with % c:

```
/*program 7.20*/
#include
main ( )
        {
          char array [7]; int I;
          printf("Type in the letters ABCDEFG:");
          scanf("%7c",array);
          for (i = size of array - 1;i >=0;i -)
              putchar (array [i]);
        }
```

Question 3: Is the array [] of program 7.20 a string?

7.5 MULTIDIMENSIONAL ARRAYS

The simplest example of a two-dimensional array is a matrix, defined as a a rectangular array of numbers. The picture is quite similar to that of rows of chairs in a classroom. Each element is accessible by two subscripts, the first referring to the row in which the element lies, the second to the column number within that row. Such a two-dimensional array is declared quite simply:

```
float matrix [5][6];
```

The declaration for matrix [ ][ ] reserves 120 bytes of storage; the first subscript, running over the rows. would vary from 0 through 4; the second from 0 through 5. The elements of a two dimensional array are stored in "row major" form. This means that they are so arranged in memory that, for minimal access time, it is the rightmost subscript that should be made to vary the, fastest. To process every element of the array matrix [ ][ ], you would need two for (;;) loops. The outer loop would run over the rows, the second over the elements within a row:

```
for (i = 0; i < 5; i ++)
  for (j = O; j < 6;j++)
    matrix [i][j] = 0.0;
```

Note that this arrangement of loops makes the rightmost subscript vary the fastest. You may

find the phrase "minimal access time" a bit strange. if you pause to consider that the CPU can retrieve any RAM location in the same amount of time; but if the array was loaded from disk (where also it is stored in row major form), it is possible that not all if it may have been "read in" in a given disk access. For large arrays such may indeed be the case in "virtual memory" machines, where disk storage may be thought of as an extension of RAM. If an array element is required that is not currently in RAM, a (very much slower) disk access would be needed to fetch it. If this happens frequently, with the CPU being forced to pause while the disk is searched, (as, for example, in processing a large two-dimensional array in column major

order) the machine will spend more time in performing disk I/O than in computing. It is then said to be "thrashing", very much like a novice swimmer splashing the water vigorously, yet unable to make any forward progress. Designers of operating systems are interested in these considerations.

The declaration:

```
static int fifty-seven [5][5] =
                {
                  19, 8, 11, 25, 7, 12, 1, 4, 18, 0,
                  16, 5, 8, 22, 4, 21, 10, 13, 27, 9,
                  14, 3, 6, 20, 2
                }
```

initializes a 5 x 5 array of ints; fifty _ seven [0][0]has the value 19; fifity _seven [1][0] is 12 and fifty seven [2][0] is 16.

Alternatively, You may declare such an array by initializing each row:

```
static int fifty-seven [ ][ ] = /* rows, cols. unspecified */

              {
                {l9,8,11,25,7},
                {12,1,4,18, 0},
                {16,5,8,22,4},
                {21,10,13,27,9},
                {14,3,6,20,2},
              }
```

This notation teaches us that a two dimensional array is actually a one dimensional array, each element of which is itself an array. Realize, therefore, that such an array can be thought of as an array of pointers. This idea is explored more fully below.

Program 7.21 reads in two matrices conformable to multiplication, and computes their product. In this program mat _ 1, a 4 x 5 matrix multiplies mat_2, a 5 x 6 matrix, and stores the result in mat_3, a 4 x 6 matrix. The [I][j]th element of mat_3 is obtained by multiplying the ith row of mat_1 into the jth column of mat_2, each element of the row to the corresponding element of the column, and then summing the several partial products.

```
/* Program 7.21 */
#include
main ( )
        {
          int i,j,k mat_1 [4][5], mat_2 [5][6], mat_3 [4][6];
          printf ("This program multiplies two matrices \n");
          prinif ("of dimensions 4 x 5 and 5 x 6.\n\n");
          printf ("Enter 20 elements for matrix # 1:");
          for (i=O;i<4;i++)
              for j = 0; j <5; j ++)
            scanf ("%d", &mat_1 [I][j]);
          printf ("\n Enter 30 elements for matrix # 2:");
          for (i = 0; i < 5; i ++)
              for ( j = 0; j <6; j ++)
            scanf ("%d", &mat_2 [I][g]);
        for (i = 0; i < 4; i ++)
              {
            for (j=O;j<6;j++)
        /* contd. */
```

```
{
            mat_3 [i][j] = 0;
            for (k=O;k < S;k++)
              mat_3 [i][j]= maL_3 [i][j] +
            mat_1 [I][k] * mat_2 [k][j];
            }
        }
    printf ("\n The given matrices are:\n\n");
    for (i = 0; i < 4; i ++)
        {
         for (j=0;j<5;j++)
           printf ("%3d ", mat_1 [i][j]);
         printf("\n");
        }
    printf ("\nand:\n\n");
    for (i = O; i<5; i ++)
        {
         for (j=0;j<6;j++)
           printf ("%3d ", mat_2 [i][j]);
         printf ("\n");
        }
    printf ("\n There product is:\n\n");
    for (i = 0; i < 4; i ++)
        {
         for (j=0;j<6;j++)
           printf ("%3d ". mat-3 [I][j]);
         printf ("\n");
        }
    }
```

A suing is a one-dimensional array: it has elements along a line. Any element can be specified by a single subscript, which gives its position in the string. If you had a bunch of strings, of equal length, each in a separate line, you would need two subscripts to specify a particular character: the first would refer to the row, e.g. the 7eroth, first, second ... string in which the char lies, the. second to its position in that string, the column number.

```
This is a two-dimensional
array of chars. It has 25
columns and 3 small rows.
```

The picture we have so far presented is for string arrays unnecessarily restrictive, because strings by their nature will be of arbitrary length. However, since strings are pointers to char, it is possible to think of a bunch of several strings as a one dimensional array of pointers. Consider the declaration:

```
static char * pntr-aray [] =
    {
      "This is the first string.",
      "This is the second string, a little longer.",
      "This is the third string.".
      "And this is the fourth and final string."
    } ;
```

First, what does this declaration mean? Is pntr aray an array of pointersto char. or is it a pointer to an array of chars? This may appear to be a confusing question but is easily answered if you reason as follows: since the subscript operator, [ ], has a higher precedence than the dereferencing operator, *(pntr_ aray []) is a

char, pntr_aray [ ] is a pointer to char, and pntr_aray is an array of pointers to char. Each element of it is a pointer to a string, because a string is a pointer to itself. The declaration of pntr-aray [] reflects this:

```
static char * pntr_aray [] =

                    {

                    pntr-1, pntr-2, pntr-3, pntr_4

                    }
```

where pntr_1, pntr_2, etc. arc pointers to the first. second, etc. strings respectively. pntr-aray [] is an array of four pointers. Program 7.22 prints these strings:

```
/* Program 7.22 */
#include
main ( )
        {
          int i,j;
          static char * pntr-aray [] =
        {
           "This is the first string.",
           "This is the second string, a little longer.',
           "This is the third string.",
           "And this is the fourth and final string."
        };
        for (i = 0; i < 4; i ++)
            {
              for (j = 0; pntr_aray [i][j], j ++)
                putchar (pntr-aray [i][j]);
             putchar ('\n');
            }
        }
```

The outer loop of Program 7.22 runs over the four pointers; the inner loop applies the subscript operator with j as subscript to extract the ith element of the ith string. (Realise that as a loop condition the expression:

```
                    pntr-aray [i][j]
```

is equivalent to:

```
                pntr-aray [I][j]!='\0'
```

The inner loop is exited when the string terminator is sensed.)

Program 7.20 creates a ragged array, in which only as much space is allocated to store the strings as is required for each separately (And for their pointers). If we had used a two dimensional array to store these strings, the row dimension would have been at least as great as to accomodate the largest string. That would wasted valuable memory, when short strings had to be stored.

Consider now the declarations
```
char ** pntr_aray;
```

Clearly, * pntr-aray is a pointer to char; when * pntr_aray is derefernced, via ** pntr_aray, it yields a char value. Then, if x is a pointer to char, * pntr_aray can be as- signed tile value of x. The implication is that

pntr-aray itself can be thought of as the address of x, a pointer. So pntr-aray declared as above can be thought as a pointer to a pointer. The contents of pntr_aray are an address. That address tells where x is stored. In x itself is another address. That address is the address of a char value. ** makes for double in-direction.

Now suppose that x, y, z, w, are a number of a string pointers, i.e. assume that they are defined thus:

```
char * x = "I am a string.";
char * y = "I am one, too.";
char * z = "I am a third string.";
char * w = "And I am the fourth and final here.";
```

Then it is possible to make the assignment

```
* pntr-aray = x.
```

Using the versatile subscript operator, another way of saying the same Pntr _ aray [0] = x;

Then one may continue:

```
pntr-aray [1] = y
pntr-aray [2] = z;
pntr-aray [3] = w;
```

Or, equally.

```
pntr_aray [0] = "I am a string.";
pntr_aray [1] = "I am one, too.";
pntr_aray [2] = "I am a third string.";
pntr_aray [3] = "And i am the fourth and final here.";
```

Thus, the declaration:

```
char ** pntr-aray;
```

implies an equivalent (though underlined) two-dimensional ragged array! An example of this usage may be seen in Program 7.23.

Check Your Progress 4

Question 1: The 5x5 array 57 has an interesting property: pick any element of each row, so that no two element lie in the same column. For every such choice, the sum of the five element chosen is always 57. Prove this by a writing a program to test for every combination.

Question 2: Rewrite program 7.21 to multiply two matrices using the pointer notation.

Question 3: Create 10x10 character arrays for each of the alphanumeric keys of the keyboard. Now write a program to print words in those large size letters, when the corresponding words are entered from the keyboard.

Question 4: Write a C program to test if a string entered from the keyboard is a palindrome, that is, that it reads the same back wards and forwards, e.g.

"Able was I saw Elba."

Question 5: Give the output of the following program:

```
/* Program 7.23*/
#include
main ( )
        {
          char**pntr-aray;
          int I,j;
          pntr_aray [0] = "I am a string.";
          pntr_aray [1] = "I am one, too.";
          pntr_aray [2] = "I am a third string.";
          pntr_aray [3} = "And I am the fourth and final here.";
          putchar(pntr_aray [0][0]);
          putchar(pntr_aray [0][1]);
          putchar(pntr_aray [2][11]);
          putchar(pntr_aray[1][5]);
          putchar(pntr_aray[0][1]);
          putchar(pntr_aray[1][6]);
          putchar(pntr_aray[1][5]);
          putchar(pntr_aray[0][8]);
          putchar(pntr_aray[0][1]);
          putchar(pntr_aray[3][13]);
          putchar(pntr_aray[0][10]);
          putchar(pntr_aray[0][11]);
        /*contd.*/

  Putchar (pntr_aray [2][11]);
          Putchar (pntr_aray [0][1]);
          Putchar (pntr_aray [2][7]);
          Putchar (pntr_aray [2][8]);
          Putchar (pntr_aray [2][9]);
          Putchar (pntr_aray [0][7]);
          Putchar (pntr_aray [0][1]);
          Putchar (pntr_aray [0][8]);
          Putchar (pntr_aray [3][14]);
          Putchar (pntr_aray [3][15]);
          Putchar (pntr_aray [2][18]);
          Putchar (pntr_aray [2][8]);
          Putchar (pntr_aray [0][13]);
        }
```

7.6 SUMMARY

In this Unit we have studied pointers and their relation to arrays. Pointers are variables that hold the addresses of other variables. They are not all alike; a pointer to char is different from a pointer to double in that the operation of incrementation causes the former to point to the next char in memory, one byte ahead; yet the same operation on a double pointer increments it by eight, the width of double. Pointers, though positive integers, therefore are not ints; neither are many of the arithmetic operations permissible for ints allowed on them. Three related operators were introduced: the "address of" operator, &, applicable to lvalues, extracts the memory address of a variable; the "contents of" operator, *, applicable to pointers, yields the value of the object addressed by a pointer; and the subscript operator, which exemplifies the intimate relationship between pointers and arrays. Applied to a pointer, this operator permits the interpretation that there exists an imaginary array with the same name as the pointer; the pointer itself holds the address of its zeroth element. Access to the "elements" of this imaginary array is enabled by a enclosing a subscript (positive, negative or zero) within the square braces of the array operator. The obverse side of

this relationship is that the name of an array is a pointer to its zeroth element Strings. being arrays of char, are pointers too; a string points to itself The key difference between a pointer declaration and an array declaration is that the former does not allocate memory to store the objects it points at. Further in this Unit, properties of the scanf () function were explored; because this function is not the most convenient for string I/O, C provides the handy functions gets () and puts (). Finally, multidimensional arrays were discussed.