UNIT 7 POINTERS AND ARRAYS

Structure **Page No.**

7.1 INTRODUCTION

This unit introduces "hard core" C: pointers and their manipulation. Pointer 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 **k**th 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"–**char**s, **float**s or **double**s, 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.

Quite apart 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 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 up to hundred million of the type $k^2 + 1$. 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.

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

In Sec. 7.2, we discuss and the operations that can be carried out using pointers. In Sec. 7.3, we discuss how pointers can be used to manipulate arrays.

`scanf()` is extremely versatile function that C uses for keyboard input. In Sec. 7.4 of this unit, we will explore some of its properties further. And finally, in Sec 7.5, 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.

Objectives

After studying this unit, you should be able to

- explain the purpose of pointers and use them;
- explain the underlying unity of pointers and arrays;
- use the subscript operator;
- exploit the versatility of `scanf()`;
- use the string I/O functions `puts()` and `gets()`; and
- 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 the address of a **char**-like variable. Similarly one can declare pointer variables to point to (i.e. hold the addresses of) **int**s, or **long**s, or **float**s or **double**s, 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 **long**s. 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 rvalues 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 question, 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 lvalue or address, fixed once and for all after compilation and linkage, and its rvalue, 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 look again 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 lvalue), 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:

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

As remarked above, the "address of" operator can only be applied to lvalues. 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 rvalue resident at a memory address. Applied to a pointer variable which has been initialised to an address, it finds the contents at that address.

In the example above, the assignment:

```
x = &z;
```

marks 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 rvalue of z, 10:

```
printf("The contents of the address held in x are:\t %d\n", *x) ;
```

See the program in Listing 1.

```
/* Program 7.1; File name: unit7-prog1.c */
#include <stdio.h>
int main ()
{
    int * x, z = 10;
    x = &z;
    printf("The address of z is :\t%u\n", x);
    printf("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);
    return (0);
}
```

Listing 1: Example to illustrate 'address of' operator and dereferencing operator.

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 program variables " "	
x, located at 56780	38790

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

Figure I

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 program variables " "	
x, located at 56780	38790

Figure II - The value of z modified indirectly

Let's now look at the program in Listing 2 on the next page. `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, c3, ..., 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 puppeteer dances her puppets. In the program in Listing 2 on the facing page, `pointer` is made to point successively at `c2, c3, ..., c9` and to make new assignments to them.

In the program in Listing 3 on the next page the pointers to **int** `p, q` and `r` are assigned the address of the **int** `x, y` and `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 value that `z` gets.

```
/* Program 7.2; File name:unit7-prog2.c */
#include <stdio.h>
int 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;
    *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",
c1, c2, c3, c4, c5, c6, c7, c8, c9);
    printf("\n");
    return (0);
}
```

Listing 2: Pointer operation.

```
/* Program 7.3; File name:unit7-prog3.c */
#include <stdio.h>
int 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);
    return (0);
}
```

Listing 3: Incrementing and decrementing pointers.

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 variable, such as for example:

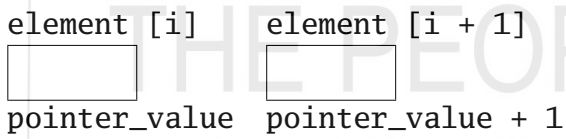
```
pointer c, x, y, p, s, t;
```

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 dereference, 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 II

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 a 32 bit PC, incrementing a pointer to **int** increases its value by 4! 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 **int**s: 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 the program in Listing 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; File name:unit7-prog4.c */
#include <stdio.h>
int main ()
{
    char *char_pointer, char_variable;
    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;
    printf("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");
    return (0);
}
```

Listing 4: Effect on incrementation of pointers of different types.

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:  3220247843, Address +
1:  3220247844
Address of int_variable:  3220247832, Address + 1:
3220247836
Address of float_variable:  3220247824, Address +
1:  3220247828
Address of double_variable:  3220247808, Address +
1:  3220247816
Moral:  Adding 1 to a pointer does not necessarily
increase it by 1.  Pointers are not ints.
```

Here are some exercises for you to try.

E1) Create, compile and execute Program 7.1, 7.2, 7.3 and 7.4.

E2) Pointers 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.

In the introduction, we mentioned the close relationship between the pointers and arrays. In the next section, we will discuss this relationship in detail.

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 **int**s. 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 address of its `(i + 1)`th element. Once can use this property in the following way: suppose you wish to scan values into the elements of an array **int** `marks [40]`. Then the following **for** (;;) loop will do:

```
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 a pointer to the `i`th 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])
```

The program in Listing 5 uses the pointer notation to store the first 1000 primes in an array, **int** `primes [1000]`.

```
/* Program 7.5; File name: unit7-prog5.c */
#define TRUE 1
#define FALSE 0
#include <stdio.h>
int main()
{
    int primes[1000], number;
    int i = 0, factor, isprime;
    *primes = 2;
    for (number = 3;; number += 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;
        }
    }
    return (0);
}
```

Listing 5: Computing first 1000 primes.

The declaration:

```
int primes [1000];
```

creates space 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 sting 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 string is in fact a pointer pointing to the byte containing 'I', execute the following simple program:

```
/* Program 7.6; File name:unit7-prog6c. */
# include <stdio.h>
int main()
{
    int i = 0;
    for (; *("I am a pointer." + i) != '\0'; i++)
        putchar(*("I am a pointer." + i));
    return (0);
}
```

Listing 6: Program showing relationship between pointers and strings.

The operation:

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

extracts the **i**th character of the string. For some value of `i` this will become the null character. At that time the conditional expression 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. The program in Listing 7 does just this. We use the %u format conversion character, since memory addresses are unsigned quantities.

```
/* Program 7.7; File name:unit7-prog7.c */
# include <stdio.h>
int main ()
{
    char * fact = "We are the best.";
    for (; * fact != '\0'; fact ++)
        printf("%c %u\n", * fact, fact);
    return (0);
}
```

Listing 7: Addresses of the characters in strings.

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 realise 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 the program in Listing 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; File name:unit7-prog8.c */
# include <stdio.h>
int 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 cntents:\n");
* fact ! = '\0'; fact ++)
        printf("\t\t%c is at %u\n", * fact, fact);
    return (0);
}
```

Listing 8: Example to illustrate memory allocation for strings.

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

```
static char string_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 Listing 5 on page 44 and Listing 6 on the previous page 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, the 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`. In as much as pointers can be 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 **i**th 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 **i**th 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; File name: unit7-prog9.c */
# include <stdio.h>
int main()
{
    int i;
    char *fact = "We are the best.";
    for (i = 0; fact[i] != '\0'; i++)
        putchar(fact[i]);
    return (0);
}
```

Listing 9: Use of subscript operator.

```
    /* 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; File name: unit7-prog10.c */
# include <stdio.h>
int main()
{
    char tree_1[20], tree_2[20], tree_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("Namea third:");
    scanf("%s", tree_3);
    printf("Name a fourth tree that you appreciate:");
    scanf("%s", tree_4);
    printf("These are your favourite trees: %s, %s, %s and %s\n",
tree_1, tree_2, tree_3, tree_4);
    printf("You\'re a good person!!!\n");
    return (0);
}
```

Here are some exercises for you.

E3) Create, compile and execute Program 7.5 – 7.10.

E4) Give the output of the programs in Listing 10 and Listing 11 on the next page.

When scanf() (with the % format conversion character) examines the stream of characters arriving from the input, it ignores all leading white space character–blanks, tabs, newlines or formfeeds; so in Program 7.10 if you responded with:

```
/* Program 7.11; File name: unit7-prog11.c */
# include <stdio.h>
```

```
int main()
{
    int i, *j;
    static int primes[] = {
2, 3, 5, 7, 11
    };
    j = primes;
    for (i = 0; i < sizeof primes / sizeof(int); i++)
        printf("%d\n", *j++);
    for (j = &primes[4]; j >= &primes[0];)
        printf("%d\n", -- *j --);
    return (0);
}
```

Listing 10: Program for exercise 4 on the previous page

```
/* Program 7.12; File name:unit7-prog12.c */
# include <stdio.h>
int 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]);
    return (0);
}
```

Listing 11: Program for exercise 4 on the previous page

<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 red 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 “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 specified number of characters is read, unless a white space character is encountered first. So, how can we read in a string with embedded space in C? We will discuss this when we discuss the `scanf()` function detail in the next section.

7.4 A DIGRESSION ON scanf ()

Executing Program 7.10 teaches us that leading white space characters in the input to `scanf`() with `%s` are ignored, while embedded white space 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 the format conversion specifier. Then every single character–including any white space characters–in the input is read. Consider:

and suppose the input to this statement was:

5m6

the 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 charcter; when it receives the input 7 it assigns it to x. With the %c format conversion charcter, 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; File name: unit-7prog13.c */
#include <stdio.h>
int 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);
    return (0);
}
```

```
/* Program 7.13:  Output */
Type in the values 5m6 for x, y and z without
space:  5m6
x = 5, y = m, z = 6
Type in the values7<SPACE>m<SPACE>8 for x, y and
z:  7 m 8
Now x = 7, y =  , z = 6
```

For clarity, we have underlined the user inputs. Programs 7.9 and 7.10 would have behaved somewhat differently if the <SPACE> 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; File name: unit7-prog14.c */
# include <stdio.h>
int main ()
{
    int x, z;
```

```
    char y;
    printf("Enter values for x, y and z, using \
spaces for separators\n");
    printf("Remeber 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("\nEnter 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);
    return (0);
}
```

```
/ * Program 7.14:  Output */
Enter values for x, y and z, using spaces for
separators
Remember to enter a non-white space 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 encounters therein a character invalid for the type listed (it was given a `'b'` when it expected a numeric value for `z`, and did not read the 56 meant for it). For decimal **int**s 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 filed width (e.g. `%4d, %6c, %3o, %2x`, etc.) may be specified: reading stops as soon as the specified number of charcters has been read in, or, as we noted in the last example, a charcter 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 space 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 terminating 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 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; File name:unit7-prog15 */
# include <stdio.h>
int main ()
{
    char string [80];
    printf("Enter a string, terminate with a tilde (~)...");
    scanf("%[^~]", string);
    printf("%s", string);
    return (0);
}
```

```
/* 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 th 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 automatically, and the "dangling" tilde is assigned to the **char** x. The call to putchar() prints the value of x.

```
/* Program 7.16; File name: unit7-pog16.c */
#include <stdio.h>
int 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);
    return (0);
}
```

```
/* Program 7.16:  Output */
Enter a string, terminate with a tilde ( )...
I am a string.
I am a string.
```

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

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() we'd used in Program 5.16:

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

The hyphens in the control string are non-format character. Their presence implied that three **int**s were to 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; File name: unit7-prog17.c */
#include <stdio.h>
int 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");
    return (0);
}
```

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 corrresponding input to be ignored. Consider the scanf():

```
scanf(''%f %*f %f'', &x, &z);
```

Suppose the input was:

12.34 45.46 78.90

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

The valid characters for an input of **float**s 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 and E and an exponent, which may be signed. The modifier l with the d or f specifiers is used to read in and assign **long**s and **double**s respectively. The modifier h with the d specifier reads **short int**s. These rules are summarised below:

%d reads **int**s in decimal notation: argument must be a pointer to **int**;
%ld reads **long int**S": argument must be a pointer to **long**;
%hd reads **short int**s: argument must be a pointer to **short**;
%u reads **unsigned int**s: argument must be a pointer to **unsigned**;
%o reads numbers in octal notation;

%lo reads **long** octal;
%ho reads **short** octals;
%x reads hex **int**s;
%lx reads **long** hex **int**s;
%hx reads short hex **int**s;
%e, \%f and %g read numers expressed in floating point notation;
%le, \%lf, \%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; File name: unit7-prog18.c */
#include <stdio.h>
int 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);
    return (0);
}
```

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 <CR> 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 **char**s is get_string[]. The <CR> 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; File name: unit7-prog19.c */
#include <stdio.h>
int main ()
{
    char get_string [55];
    puts ("Tell me how you are");
    puts ("in 55 keystrokes or less.");
    puts ("That\'s inclusive of the <CR>,");
    puts ("which we expect you to type presently!");
    printf("%s", "Begin here: ");
    gets (get_string);
    puts ("Thanks for that detailed description.");
    return (0);
}
```

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.

E5) Describe what happens if you input more than 55 characters to Program 7.19.

E6) Give the output of the following program, in which a field width is specified with %c:

```
/* Program 7.20; File name:unit7-prog20.c */
#include<stdio.h>
int main ()
{
    char array [7]; int i;
```

```
        printf("Type in the letters ABCDEFG: ");
        scanf("%7c", array);
        for (i = sizeof array - 1;i >= 0; i --)
            putchar (array [i]);
        return (0);
    }
```

E7) Is the array [] of Program 7.20 a string?

We close this section here. We will discuss multidimensional arrays in the next section.

7.5 MULTIDIMENSIONAL ARRAYS

The simplest example of a two-dimensional array is a matrix, defined as 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 element 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 faster. 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 = 0; 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 times; 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
};
```

initialises a 5×5 array of **int**s; `fifty_seven [0][0]` has the value 19, `fifty_seven [1][0]` is 12 and `fifty_seven [2][0]` is 16.

Alternatively, you may declare such an array by initialising each row:

```
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}
}
```

This notation teaches us that a two dimensional array is actually a one dimensional array, each element of which is itself an array. Realise, 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 4×5 matrix multiplies `mat_2`, a 5×6 matrix, and stores the result in `mat_3`, a 4×6 matrix. The `[i][j]`th element of `mat_3` is obtained by multiplying the **ith** row of `mat_1` into the **j**th 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; File name: unit7-prog21.c */
#include <stdio.h>
int main()
{
    int i, j, k, mat_1[4][5], mat_2[5][6], mat_3[4][6];
    printf("This program multiplies two matrixes\n");
    printf("of dimensions 4 x 5 and 5 x 6.\n\n");
    printf("Enter 20 elements for matrix # 1: ");
    for (i = 0; i < 4; i++)
        for (j = 0; j < 5; j++)
            scanf("%d", &mat_1[i][j]);
    printf("\nEnter 30 elements for matrix # 2: ");
    for (i = 0; i < 5; i++)
        for (j = 0; j < 6; j++)
            scanf("%d", &mat_2[i][j]);
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 6; j++) {
            mat_3[i][j] = 0;
            for (k = 0; k < 5; k++)
                mat_3[i][j] = mat_3[i][j] +
mat_1[i][k] * mat_2[k][j];
        }
    }
    printf("\nThe given matrixes 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 = 0; i < 5; i++) {
        for (j = 0; j < 6; j++)
            printf("%3d", mat_2[i][j]);
        printf("\n");
    }
    printf("\nTheir product is:\n\n");
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 6; j++)
            printf("%3d", mat_3[i][j]);
        printf("\n");
    }
```

```
    return (0);
}
```

A string 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 zeroth, 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 pointers to **char**, or is it a pointer to an array of **char**s? 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, are pointers to the first, second, etc. strings respectively. `pntr_array []` is an array of four pointers. The program Listing 12 on the facing page prints these strings: 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 `j`th element of the `i`th string. (Realise that as a loop condition the expression:

```
pntr_aray [i][j]
```

is equivalent to:

```
pntr_array [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 accommodate the largest string. That would have wasted valuable memory when short strings had to be stored.

Consider now the declaration:

```
/* Program 7.22: File name: unit7-prog22.c */
#include <stdio.h>
int 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]; ++)
            putchar((pntr_aray)[i][j]);
        putchar("\n");
    }
    return (0);

}
```

Listing 12: Array of strings as two dimensional arrays

```
char ** pntr_aray;
```

Clearly, * pntr_aray is a pointer to **char**; when * pntr_aray is dereferenced, via ** pntr_aray, it yields a **char** value. Then, if x is a pointer to **char**, * pntr_aray can be assigned the 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. ** marks for double indirection.

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 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 thing as:

```
pntr_aray [0] = x;
```

The 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 undefined) two-dimensional ragged array! An example of this usage may be seen in Program 7.23.

E8) Rewrite Program 7.21 to multiply two matrices using the pointer notation.

E9) Write a C program to test if a string entered from the keyboard is a palindrome, that is, that it reads the same backwards and forwards, e.g.

"Able was I ere I saw Elba."

E10) Give the output of the following program:

```
/* Program 7.23; File name: unit7-prog23.c */
#include <stdio.h>
int main ()
{
        char ** pntr_aray;
        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]);
        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]);
        return (0);
}
```

We have now completed the unit. You may like go through the summary given in the next section to recaptitulate what we have studied so far.

7.6 SUMMARY

In this unit we have studied the following:

1. Pointers are **int** like variables that hold the addresses of other variables.
2. The rules of pointer arithmetic: The incrementation of pointers is different from the incrementation of other variables. For example, depending on whether it points to **int**, **char**, on incrementation it points to 8 bytes or one byte ahead.

3. Three operators are used in working with pointers, the "address of " of operator &, the "contents of" operator * and the subscript operator `[]`.

4. The name of a array is a pointer to its zeroth element. So, all the rules of pointer arithmetic apply to array name.

5. The format conversion rules followed by `scanf()`.

%d	Reads **int**s in decimal notation. The argument must be a pointer to **int**.
%i	Reads an integer with a prefix or suffix. Prefixes allowed are a sign(+ or −) or `o` to denote an octal constant, or one of `ox`, `oX` to denote a hex constant. Suffixes allowed are `u` or `U` to denote an **unsigned int** and `l` or `L` to denote a **long int**.
%ld	Reads **long int**s; the argument should be a pointer to **long**.
%hd	Reads **short int**s; argument must be a pointer to **unsigned**.
%u	Reads **unsigned int**s; argument must a pointer to **unsigned**.
%o	Reads numbers octal notation.
%lo	Reads **long** octals.
%ho	Reads **short** octals.
%x, %X	Reads hex `its`s.
%lx, %lX	Reads **long** hex **int**s.
%hx, %hX	Reads **short** hex **int**s.
%e, %E, %f, %f and %G	is for reading numbers expressed in floating point notation.
%le, %lE, %lf, %lf and %lG	is for reading **double**. The arguments should be pointers to **double**.
%Le, %LE, %Lf, %Lf and %LG	The arguemts must be pointers to **long double**.
%c	Reads single characters. White space characters are significant. Therefore, to read the next non-space character, use `%ls`. Argument corresponding to `%c` is a pointer to a **char**.
%s	Used for reading a character string. Characters are read until a white space character is encountered. The corresponding argument is a an array of **char**s, which must be big enough to hold the characters entered, plus the null character which `scanf()` appends.
%p	Reads a pointer. The argument expected is a pointer to **void**.
%n	records the number ofcharacters read so far in this call to `scanf()`; while no characters are read from the input stream against \%n, its corresponding data argument is a pointer to **int**.

%[...]	Reads a string until a character that is not found in the square braces is encountered. If the first character in the braces is ^, then the subsequent characters listed there serve as terminators. The input stream is read until one of the characters listed after the caret is encountered.
%%	Matches a single percent sign % in the input stream, without any assignment being made.

6. We discussed multidimensional arrays and ragged arrays in which different rows are allowed to have different length.

7.7 SOLUTIONS/ANSWERS

E2) Add the line

```
printf("The address of x is %u",&x);
```

before the **return** `(0);` statement.

E5) The program will crash. This is because there is no memory available and the program will try to write to a memory location not allocated to the program.

E8) The program is given in Listing 13. Note the second * in the statements

```
*(*(mat_3+i)+j)= 0;
```

for accessing the value `mat_3[i][j]`.

```
/* Program 7.21; File name: unit7-prog21.c */
#include <stdio.h>
int main()
{
    int i, j, k, mat_1[4][5], mat_2[5][6], mat_3[4][6];
    printf("This program multiplies two matrixes\n");
    printf("of dimensions 4 x 5 and 5 x 6.\n\n");
    printf("Enter 20 elements for matrix # 1: ");
    for (i = 0; i < 4; i++)
        for (j = 0; j < 5; j++)
            scanf("%d", (*(mat_1+i)+j));
    printf("\nEnter 30 elements for matrix # 2: ");
    for (i = 0; i < 5; i++)
        for (j = 0; j < 6; j++)
            scanf("%d", (*(mat_2+i)+j));
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 6; j++) {
            *(*(mat_3+i)+j)= 0;
            for (k = 0; k < 5; k++)
                *(*(mat_3+i)+j) = *(*(mat_3+i)+j) +
*(*(mat_1+i)+k) *  *(*(mat_2+ k)+j);
        }
    }
    printf("\nThe given matrixes 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 = 0; i < 5; i++) {
        for (j = 0; j < 6; j++)
            printf("%3d", mat_2[i][j]);
        printf("\n");
    }
    printf("\nTheir product is:\n\n");
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 6; j++)
            printf("%3d", mat_3[i][j]);
        printf("\n");
    }
    return (0);
}
```

Listing 13: Solution to exercise 8.

E9) See Listing 14 for the program:

```
#include<stdio.h>
#define NUM_CHAR 40
int main()
{
    int i = -1,k;
    char c, input[NUM_CHAR];
    printf("Enter a string of length atmost %d\n"
    , NUM_CHAR);
    printf("Terminate your string with <CR>:\n");
    printf("All your characters must be in lower case.\n");
    while ((c = getchar()) != '\n'){
        i++;
        input[i]=c;
    }
    for (k = 0; k <= i/2; k++)
        if (input[i-k] != input[k]){
            break;
        }
    if (k < i/2)
        printf("The string is not a palindrome.");
    else
        printf("\n The string is a palindrome.");
    return (0);
}
```

Listing 14: Solution to exercise 9.