Unit 9 Pointers – II

Structure:

9.1 Introduction

In the previous unit, you read about the basics of pointers. A pointer is a variable that points at, or refers to, another variable. Pointers are very useful when you want to refer to some other variable by pointing at it. Pointers not only point to single variables but can also point at the cells of an array. Pointers provide a convenient way to represent multidimensional arrays, allowing a single multidimensional array to be replaced by a lower-dimensional array of pointers. This feature permits a collection of strings to be represented within a single array, even though the individual strings may differ in length.

In this unit, you will read about null pointers and how pointers can be used to manipulate a string along with the usage of pointers in two dimensional arrays. You will also read about arrays of pointers briefly.

Objectives:

After studying this unit, you should be able to:

- explain a null pointer
- use pointers as function arguments ?
- manipulate strings by the help of pointers
- explain arrays of pointers

9.2 Null Pointers

We said that the value of a pointer variable is a pointer to some other variable. There is one other value a pointer may have: it may be set to a *null pointer*. A *null pointer* is a special pointer value that is known not to point anywhere. What this means that no other valid pointer, to any other variable or array cell or anything else, will ever compare equal to a null pointer.

The most straightforward way to “get" a null pointer in your program is by using the predefined constant NULL, which is defined for you by several standard header files, including <stdio.h>, <stdlib.h>, and <string.h>. To initialize a pointer to a null pointer, you might use code like

```
#include <stdio.h>
int *ip = NULL;
```

and to test it for a null pointer before inspecting the value pointed to, you might use code like

```
if(ip != NULL)
        printf("%d\n", *ip);
```

It is also possible to refer to the null pointer by using a constant 0, and you will see some code that sets null pointers by simply doing

```
int *ip = 0;
```

(In fact, NULL is a preprocessor macro which typically has the value, or replacement text, 0.)

Furthermore, since the definition of “true" in C is a value that is not equal to 0, you will see code that tests for non-null pointers with abbreviated code like

```
if(ip)
        printf("%d\n", *ip);
```

This has the same meaning as our previous example; if(ip) is equivalent to if(ip != 0) and to if(ip != NULL).

All of these uses are legal, and although the use of the constant NULL is recommended for clarity, you will come across the other forms, so you should be able to recognize them.

You can use a null pointer as a placeholder to remind yourself (or, more importantly, to help your program remember) that a pointer variable does not point anywhere at the moment and that you should not use the “contents of'' operator on it (that is, you should not try to inspect what it points to, since it doesn't point to anything). A function that returns pointer values can return a null pointer when it is unable to perform its task. (A null pointer used in this way is analogous to the EOF value that functions like getchar return.)

As an example, let us write our own version of the standard library function strstr, which looks for one string within another, returning a pointer to the string if it can, or a null pointer if it cannot.

Example 9.1: Here is the function, using the obvious brute-force algorithm: at every character of the input string, the code checks for a match there of the pattern string:

```
#include <stddef.h>

char *mystrstr(char input[], char pat[])
{
        char *start, *p1, *p2;
        for(start = &input[0]; *start != '\0'; start++)
                {               /* for each position in input string... */
                p1 = pat;       /* prepare to check for pattern string there */
                p2 = start;
                while(*p1 != '\0')
                        {
                        if(*p1 != *p2)   /* characters differ */
                                break;
                        p1++;
                        p2++;
                        }
                if(*p1 == '\0')             /* found match */
                        return start;
                }
        return NULL;
}
```

The start pointer steps over each character position in the input string. At each character, the inner loop checks for a match there, by using p1 to step over the pattern string (pat), and p2 to step over the input string (starting at start). We compare successive characters until either (a) we reach the end of the pattern string (*p1 == '\0'), or (b) we find two characters which differ. When we're done with the inner loop, if we reached the end of the pattern string (*p1 == '\0'), it means that all preceding characters matched, and we found a complete match for the pattern starting at start, so we return start. Otherwise, we go around the outer loop again, to try another starting position. If we run out of those (if *start == '\0'), without finding a match, we return a null pointer.

Notice that the function is declared as returning (and does in fact return) a pointer-to-char.

In general, C does not initialize pointers to null for you, and it never tests pointers to see if they are null before using them. If one of the pointers in your programs points somewhere some of the time but not all of the time, an excellent convention to use is to set it to a null pointer when it doesn't point anywhere valid, and to test to see if it's a null pointer before using it. But you must use explicit code to set it to NULL, and to test it against NULL. (In other words, just setting an unused pointer variable to NULL doesn't guarantee safety; you also have to check for the null value before using the pointer.) On the other hand, if you know that a particular pointer variable is always valid, you don't have to insert a paranoid test against NULL before using it.

Self Assessment Questions

1. A _____ is a special pointer value that is known not to point anywhere.
2. A function that returns _____ values can return a null pointer when it is unable to perform its task.
3. In general, C does not initialize pointers to null for you, and it never tests pointers to see if they are null before using them. (True/False)

9.3 Pointers as Function Arguments

Earlier, we learned that functions in C receive copies of their arguments. (This means that C uses *call by value*; it means that a function can modify its arguments without modifying the value in the caller.)

Consider the following function to swap two integers

```
void swap(int x, int y)
{
        int temp;
        temp=x;
        x=y;
        y=temp;
        return;
}
```

The problem with this function is that since C uses *call by value* technique of parameter passing, the caller can not get the changes done in the function. This can be achieved using pointers as illustrated in the following program.

Program 9.2: Program to swap two integers using pointers

```
#include<stdio.h>
main()
{
        int a, b;
        void swap(int *a, int *b);
        printf(" Read the integers:");
        scanf("%d%d", &a, &b);
        swap(&a, &b);           /* call by reference or call by address*/
        printf(" \nAfter swapping:a=%d        b=%d", a, b);
}
void swap(int *x, int *y)
{
        int temp;
        temp=*x;
        *x=*y;
        *y=temp;
        return;
}
```

Execute this program and observe the result.

Because of the use of call by reference, the changes made in the function swap() are also available in the main().

9.4 Pointers and Strings

Because of the similarity of arrays and pointers, it is extremely common to refer to and manipulate strings as character pointers, or char *'s. It is so common, in fact, that it is easy to forget that strings are arrays, and to imagine that they're represented by pointers. (Actually, in the case of strings, it may not even matter that much if the distinction gets a little blurred; there's certainly nothing wrong with referring to a character pointer, suitably initialized, as a ’’string.") Let's look at a few of the implications:

Any function that manipulates a string will actually accept it as a char * argument. The caller may pass an array containing a string, but the function will receive a pointer to the array's (string's) first element (character).

The %s format in printf expects a character pointer.

Although you have to use strcpy to copy a string from one array to another, you can use simple pointer assignment to assign a string to a pointer. The string being assigned might either be in an array or pointed to by another pointer. In other words, given

```
char string[] = "Hello, world!";
char *p1, *p2;
```

both

```
p1 = string
```

and

```
p2 = p1
```

are legal. (Remember, though, that when you assign a pointer, you're making a copy of the pointer but *not* of the data it points to. In the first example, p1 ends up pointing to the string in string. In the second example, p2 ends up pointing to the same string as p1. In any case, after a pointer assignment, if you ever change the string (or other data) pointed to, the change is “visible" to both pointers.

Many programs manipulate strings exclusively using character pointers, never explicitly declaring any actual arrays. As long as these programs are careful to allocate appropriate memory for the strings, they're perfectly valid and correct.

When you start working heavily with strings, however, you have to be aware of one subtle fact.

When you initialize a character array with a string constant:

```
char string[] = "Hello, world!";
```

you end up with an array containing the string, and you can modify the array's contents to your heart's content:

```
string[0] = 'J';
```

However, it's possible to use string constants (the formal term is *string literals*) at other places in your code. Since they're arrays, the compiler generates pointers to their first elements when they're used in expressions, as usual. That is, if you say

```
char *p1 = "Hello";
int len = strlen("world");
```

it's almost as if you'd said

```
char internal_string_1[] = "Hello";
char internal_string_2[] = "world";
char *p1 = &internal_string_1[0];
int len = strlen(&internal_string_2[0]);
```

Here, the arrays named internal_string_1 and internal_string_2 are supposed to suggest the fact that the compiler is actually generating little temporary arrays every time you use a string constant in your code. *However*, the subtle fact is that the arrays which are “behind" the string constants are *not* necessarily modifiable. In particular, the compiler may store them in read-only-memory. Therefore, if you write

```
char *p3 = "Hello, world!";
p3[0] = 'J';
```

your program may crash, because it may try to store a value (in this case, the character 'J') into nonwritable memory.

The moral is that whenever you're building or modifying strings, you have to make sure that the memory you're building or modifying them in is writable. That memory should either be an array you've allocated, or some memory which you've dynamically allocated. Make sure that no part of your program will ever try to modify a string which is actually one of the unnamed,

unwritable arrays which the compiler generated for you in response to one of your string constants. (The only exception is array initialization, because if you write to such an array, you're writing to the array, not to the string literal which you used to initialize the array.)

Example 9.3: Breaking a Line into “Words"
First, break lines into a series of whitespace-separated words. To do this, we will use an *array of pointers to* char, which we can also think of as an “array of strings," since a string is an array of char, and a pointer-to-char can easily point at a string. Here is the declaration of such an array:

```
char *words[10];
```

This is the first complicated C declaration we've seen: it says that words is an array of 10 pointers to char. We're going to write a function, getwords, which we can call like this:

```
int nwords;
nwords = getwords(line, words, 10);
```

where line is the line we're breaking into words, words is the array to be filled in with the (pointers to the) words, and nwords (the return value from getwords) is the number of words which the function finds. (As with getline, we tell the function the size of the array so that if the line should happen to contain more words than that, it won't overflow the array).

Here is the definition of the getwords function. It finds the beginning of each word, places a pointer to it in the array, finds the end of that word (which is signified by at least one whitespace character) and terminates the word by placing a '\0' character after it. (The '\0' character will overwrite the first whitespace character following the word.) Note that the original input string is therefore modified by getwords: if you were to try to print the input line after calling getwords, it would appear to contain only its first word (because of the first inserted '\0').

```
#include <stddef.h>
#include <ctype.h>

getwords(char *line, char *words[], int maxwords)
{
char *p = line;
```

```
int nwords = 0;

while(1)
        {
        while(isspace(*p))
                p++;

        if(*p == '\0')
                return nwords;
        words[nwords++] = p;

        while(!isspace(*p) && *p != '\0')
                p++;

        if(*p == '\0')
                return nwords;

        *p++ = '\0';

        if(nwords >= maxwords)
                return nwords;
        }
}
```

Each time through the outer while loop, the function tries to find another word. First it skips over whitespace (which might be leading spaces on the line, or the space(s) separating this word from the previous one). The isspace function is new: it's in the standard library, declared in the header file <ctype.h>, and it returns nonzero (“true") if the character you hand it is a space character (a space or a tab, or any other whitespace character there might happen to be).

When the function finds a non-whitespace character, it has found the beginning of another word, so it places the pointer to that character in the next cell of the words array. Then it steps through the word, looking at non-whitespace characters, until it finds another whitespace character, or the \0 at the end of the line. If it finds the \0, it's done with the entire line; otherwise,

it changes the whitespace character to a \0, to terminate the word it's just found, and continues. (If it's found as many words as will fit in the words array, it returns prematurely.)

Each time it finds a word, the function increments the number of words (nwords) it has found. Since arrays in C start at [0], the number of words the function has found so far is also the index of the cell in the words array where the next word should be stored. The function actually assigns the next word and increments nwords in one expression:

```
words[nwords++] = p;
```

You should convince yourself that this arrangement works, and that (in this case) the preincrement form

```
words[++nwords] = p;          /* WRONG */
```

would *not* behave as desired.

When the function is done (when it finds the \0 terminating the input line, or when it runs out of cells in the words array) it returns the number of words it has found.

Here is a complete example of calling getwords:

```
char line[] = "this is a test";
int i;
nwords = getwords(line, words, 10);
for(i = 0; i < nwords; i++)
        printf("%s\n", words[i]);
```

Self Assessment Questions

4. An array of characters is called as _____.
5. You can represent an array of strings using pointers by using an array of _____ to character.
6. A string must always be terminated with a _____ character.

9.5 Pointers and two-dimensional arrays

Since a one-dimensional array can be represented in terms of pointer (the array name) and an offset (the subscript), it is reasonable to expect that a two-dimensional array can also be represented with an equivalent pointer

notation. A two-dimensional array is actually a collection of one-dimensional arrays. Therefore we can define a two dimensional array as a pointer to a group of contiguous one-dimensional arrays. A two-dimensional array declaration can be written as:

*data-type (*ptr)[expression]*

where *data-type* is the type of array elements and *expression* is the positive-valued integer expression that indicates the number of columns in each row of the two-dimensional array.

Example 9.5: Suppose that x is a two-dimensional integer array having 10 rows and 20 columns. We can declare x as

```
int (*x)[20];
```

In this case, x is defined to be a pointer to a group of contiguous, one-dimensional, 20-element integer arrays. Thus x points to the first 20-element array, which is actually the first row(row 0) of the original two-dimensional array. Similarly, (x+1) points to the second 20-element array, which is the second row(row 1) of the original two-dimensional array and so on.

Arrays of pointers

A two-dimensional array can also be expressed in terms of an array of pointers rather than as a pointer to a group of contiguous arrays. Each element in the array is a pointer to a separate row in the two-dimensional array. In general terms, a two-dimensional array can be defined as a one-dimensional array of pointers by writing:

*data-type *ptr[expression]*

where *data-type* is the type of array elements and *expression* is the positive-valued integer expression that indicates the number of rows.

Example 9.6: Suppose that x is a two-dimensional array having 10 rows and 20 columns, we can define x as a one-dimensional array of pointers by writing

```
int *x[10];
```

Hence, x[0] points to the beginning of the first row, x[1] points to the beginning of the second row and so on.

An individual array element, such as x[2][4] can be accessed

by writing

*(x[2]+4)

In this expression, x[2] is a pointer to the first element in row 2, so that (x[2]+5) points to element 4(actually, the fifth element) within row 2. The element of this pointer, *(x[2]+4), therefore refers to x[2][4].

Self Assessment Questions

7. We can define a two dimensional array as a pointer to a group of contiguous _____ dimensional arrays.
8. A two-dimensional array can also be expressed in terms of an _____ of pointers rather than as a pointer to a group of contiguous arrays.

9.6 Summary

A pointer is a variable that represents the location of a data item, such as a variable or an array element. A pointer may be set to a null pointer. A null pointer is a special pointer value that is known not to point anywhere. Passing pointers as arguments to functions is called pass by reference. Because of the similarity of arrays and pointers, it is extremely common to refer to and manipulate strings as character pointers. We can define a two dimensional array as a pointer to a group of contiguous one-dimensional arrays. A two-dimensional array can also be expressed in terms of an array of pointers rather than as a pointer to a group of contiguous arrays.

9.7 Terminal Questions

1. Explain what a null pointer is. Illustrate it with an example.
2. How can you use pointers as function arguments? Illustrate it with an example.
3. Distinguish between pass by value and pass by reference with the help of an example.
4. How can you manipulate strings using character pointers? Illustrate it with the help of an example.

9.8 Answers to Self Assessment Questions

1. null pointer
2. pointer
3. True
4. string
5. pointers
6. null
7. one
8. array

9.9 Answers to Terminal Questions

1. A null pointer is a special pointer value that is known not to point anywhere and is implemented by using the predefined constant NULL, which is defined by several standard header files, including <stdio.h>, <stdlib.h>, and <string.h>. (Refer to section 9.2 for more details)
2. (Refer to section 9.2 for more details)
3. In pass by value technique, the actual value is passed to the function whereas in pass by reference, the address is passed to the function.
4. You can use the pointer to the string considering string as an array and use (string's) first element i.e. character. The %s format in printf expects a character pointer. (Refer to section 9.4 and 9.5 for more details)

9.10 Exercises

1. Write a program to find the number of characters in a string using pointers.
2. Write a program to multiply two matrices using pointers.
3. Suppose a formal argument within a function definition is a pointer to another function. How is the formal argument declared? Within the formal argument declaration, to what does the data type refer?
4. Write a program to concatenate two strings.
5. Write a function to sort an array of numbers using pointers.