7. POINTERS

An often more convenient and efficient way to access a variable is through a second variable that holds the address of the variable to be accessed. In this chapter, we will examine one of the most sophisticated features of the C programming language: **pointers**. In fact, the power and flexibility that C provides in dealing with pointers serve to set it apart from other programming languages such as Pascal. Pointers enable you to effectively represent complex data structures, to change values passed as arguments to functions, to work with memory that has been allocated "**dynamically**" and to more concisely and efficiently deal with arrays.

To understand the way in which pointers operate, it is first necessary to understand the concept if ***indirection***. We are used this concept in our every day life. For example, suppose that I needed to buy a new ribbon for my printer. In the company that I work for, all purchases are handled by the purchasing department. So I would call Jim in purchasing and ask him to order the new ribbon. The approach that I would take in obtaining my new ribbon would actually be an indirect one, since I would not be ordering the ribbon directly from the supply store myself.

This same notion of indirection applies to the way pointers work in C. A pointer provides an indirect means of accessing the value of a particular data item. And just as there are reasons why it makes sense to go through the purchasing department to order new ribbons (I don't have to know which particular store the ribbons are being ordered from, for example), so are there good reasons why, at times, it makes sense to use pointers in C.

But enough talk-it's time to see how pointers actually work. Suppose we define a variable called ***count*** as follows:

```
int  count  =  10 ;
```

We can define another variable, called ***int_pointer***, that will enable us to indirectly access the value of ***count*** by the declaration.

```
int  *int_pointer ;
```

The asterisk defines to the C system that the variable ***int_pointer*** is of type *pointer* to *int*. This means that ***int_pointer*** will be used in the program to indirectly access the value of one or more integer variables.

We have seen how the & operator was used in the ***scanf*** calls of previous programs. This unary operator, known as the *address* operator, is used to make a pointer to an object in C. So if x is a variable of a particular type, then the expression &x is a pointer to that variable. The expression &x can be assigned to any pointer variable, if desired, that has been declared to be a pointer to the same type as x.

Therefore, with the definitions of *count* and ***int_pointer*** as given, we can write a statement such as

```
int_pointer  =  &count ;
```

to set up the indirect reference between ***int_pointer*** and ***count***. The address operator has the effect of assigning to the variable ***int_pointer***, not the value of *count*, but a pointer to the variable ***count***. The link that has been made between ***int_pointer*** and ***count*** is conceptualized in below figure. The directed line illustrates the idea that ***int_pointer*** does not directly contain the value of *count*, but a pointer to the variable *count*.

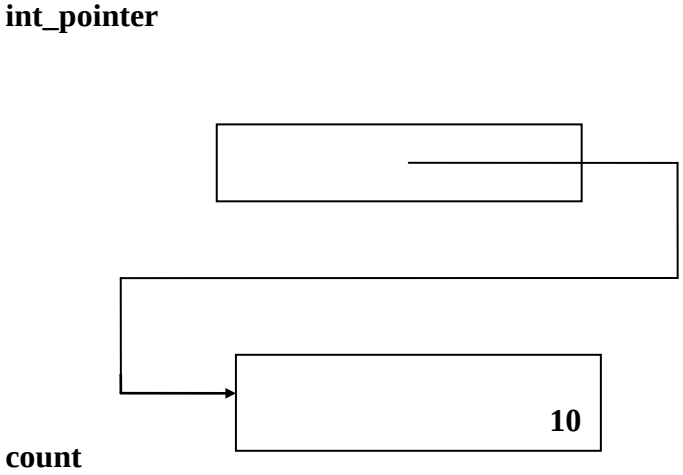


Figure : Pointer to an integer

In order to reference the contents of *count* through the pointer variable ***int_pointer***, we use the ***indirection*** operator, which is the asterisk ∗. So if x were defined to be of type *int*, then the statement

```
x = ∗int_pointer ;
```

would assign the value that is indirectly referenced through ***int_pointer*** to the variable x. Since ***int_pointer*** was previously set pointing to *count*, this statement would have the effect of assigning the value contained in the variable *count*-which is 10-to the variable x.

The previous statements have been incorporated into the following program, which illustrates the two fundamental pointer operators: the address operator, &, and the indirection operator, ∗.

7.1 Declaring Pointer Variables

As with any other language, C requires a definition for each variable. The following statement defines a pointer variable ***px*** that can hold the address of an **int** variable:

```
int ∗px ;
```

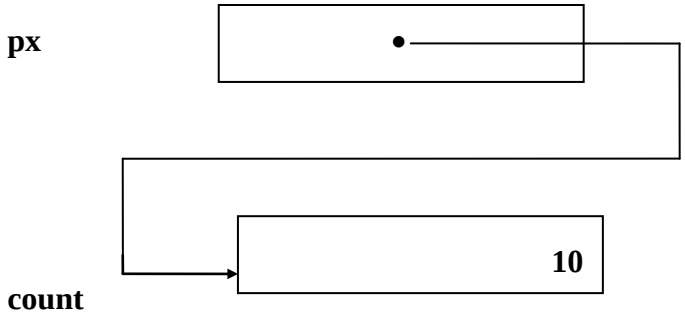


Figure : Assignment using a Pointer variable

Actually, there are two separate parts to this declaration. The data type of px is

```
int *
```

and the identifier for the variable is

```
px
```

The asterisk following **int** means "**pointer to**"; that is, the data type

```
int *
```

is a pointer variable that can hold an address to an **int**.

This is a very important concept to remember. In C, unlike many other languages, a pointer variable holds the address of a particular data type. Here's an example:

```
char *address_to_a_char ;

int  *address_to_ an_int ;
```

The data type of ***address_to_a_char*** is different from the type of ***address_to_an_int***. Run-time errors and compile-time warnings may occur in a program that defines a pointer to one data type and then uses it to point to some other data type. It is also poor programming practice to define a pointer in one way and then use it in another way. For example, look at the following code segment:

```
int *int_ptr ;

float real_value = 23.45 ;
```

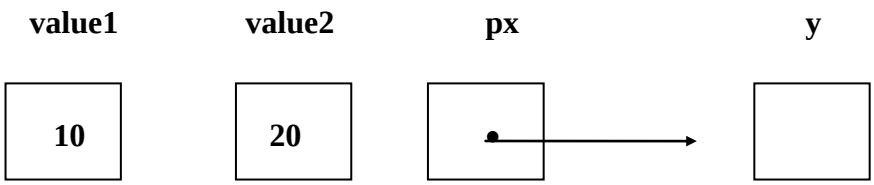
```
int_ptr = &real_value ;
```

Here, ***int_ptr*** has been defined to be of type **int ∗**, meaning that it can hold the address of a memory cell of type **int**. The third statement attempts to assign ***int_ptr*** the address ***&real_value*** of a declared **float** variable.

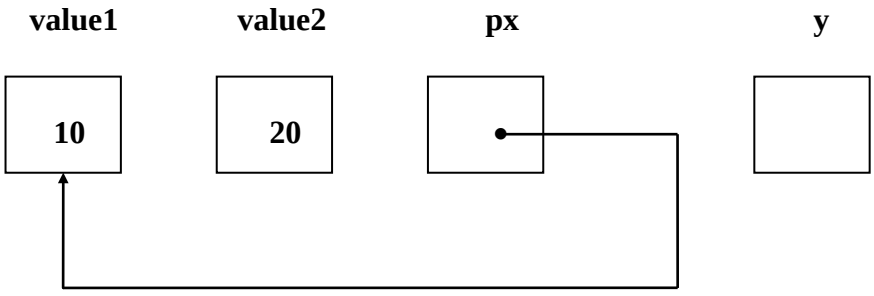
7.2 Using Pointer Variables

The following code segment will exchange the contents of the variables value1 and value2 by using the address and **dereferencing operators**:

```
int   value1 = 10 ,  value2 = 20 ,  y  ,  ∗px ;
```




```
px  =  &value1 ;
```



y = ∗px ; ⇒ y is assigned the value pointed by px since px points to value1.

y = ∗px ⇒ y = value1 ⇒ y = 10

7.3 Initializing Pointer Variables

Pointer variables, like many other variables in C, can be initialized in their definition. For example, the following two statements

```
int  value1 ;

int  *int_ptr  =  value1 ;
```

allocate storage for the two cells ***value1*** and ***int_ptr***. The variable ***value1*** is an ordinary **int** variable and ***int_ptr*** is a pointer to an **int**. Additionally, the code initializes the pointer variable ***int_ptr*** to the address of ***value1***. Be careful, however; the syntax is somewhat misleading. You are not initializes ****int_ptr*** (which would have to be an **int** value) but ***int_ptr*** (which must be an address to an **int**). The second statement in the previous listing can be translated into the equivalent two statements:

```
int  value1 ;

int  *int_ptr  =  &value1 ;
```

The following code segment shows how to declare and then initialize a string pointer:

```
# include <stdio.h>
# include <string.h>

void main(void)
{

        char  *p  =  "University of Colombo" ;
        int  index ;

        for( index = strlen(p) - 1  ;  index >= 0  ;  index-- )

                printf("%c" , p[index] ) ;

}
```

Technically, the compiler stores the address of the first character of the string "**University of Colombo**" in the variable *p*. While the program is running, it can use p like any other string. This is because all C compilers create what is called a string table, which is used internally by the compiler to store the string constants a program is using.

Eg :

```
# include <stdio.h>

void main(void )
{
        char  *p ;

        char Line[ ]  =  "Hello there world" ;

        p = &Line[0] ;                  ⇒    p points to address of Line[0]

        printf("%s\n" , p ) ;           ⇒    outputs  "Hello there world"

        p = &Line[6] ;                  ⇒    p points to address of Line[6]

        printf("%s\n" , p ) ;           ⇒    outputs  "there world"

        *p = 'w' ;                      ⇒    t  will be replaced with w

        printf("%s\n" , p ) ;           ⇒    outputs  "where world"

        printf("%s\n", Line) ;          ⇒    outputs  "Hello where world"
```

using '**strcpy**' can change the characters from 'p' onwords.

```
        strcpy( p , "goodbye" ) ;
```

overwrites the contents of '**Line**' from Line[6] onwords.

```
        printf("%s\n", Line) ;          ⇒    outputs  "Hello goodbye"
```

Eg :

```
# include <stdio.h>
# include <string.h>
# define  MAX  100

void main(void)
{
        char C = 'a' ,  *p ,  S[MAX] ;

        p = &C ;

        printf("%c%c%c\n", *p , *p+1 , *p+2 ) ;

        strcpy( S , "ABC" ) ;
```

```
        printf("%s%c%c%s\n", S , *S+6 , *S+7 , S+1 ) ;

        strcpy( S , "she sells sea shells by the seashore" ) ;

        p = S + 14 ;

        for( ; *p != '\0' ; ++p )
        {
                if( *p == 'e' )

                        *p = 'E' ;

                if( *p == ' ' )

                        *p = '\n' ;
        }

        printf("%s\n" , S ) ;

}
```

The output of this program:

```
abc
ABCGHBC
she sells sea shElls
by
thE
sEashorE
```

7.4 Limitations on the Address Operator

You cannot use the address operator on every C expression. The following examples demonstrate those situations in which you cannot apply the **&** address operator:

```
variable_address = &23 ;                /* not with constants */

variable_address = &(value1 + 10) ;     /* not with expressions involving
                                           operators such as + and given the
                                           definition int value1 = 8 ;  */

variable_address = &reg1 ;              /* not preceding register variables
                                           given the definition register reg1;  */
```

The first statement tries to obtain the address of a hard-wired constant value illegally. Since the 23 has no memory cell associated with it, the statement is meaningless.

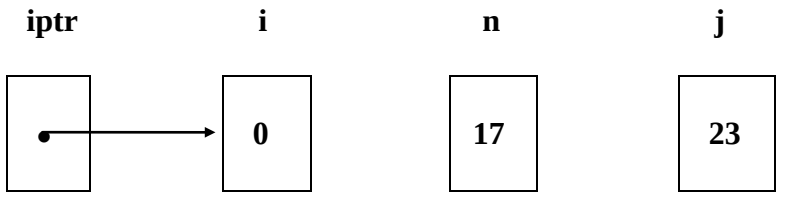
The second assignment statement attempts to return the address of the expression (value1 + 10). Since the expression itself is actually a stack manipulation process, there is no address associated with the expression.

The last example honors the programmer's request to define reg1 as a register rather than as a storage cell in internal memory. Therefore, no memory cell address can be returned and stored.

7.5 Dereferencing Pointer Variables

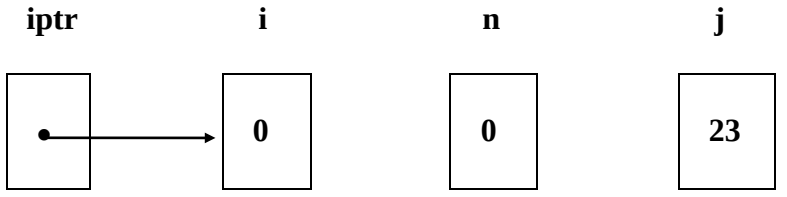
Once we have made a pointer point to something, we can access the value to which it points, a process called **dereferencing**. Dereferencing is done with the indirection operator ∗, which, when executed, follows the pointer and returns the value at that address. In the example below, **∗iptr** is 0, the value of i.

(**a**)



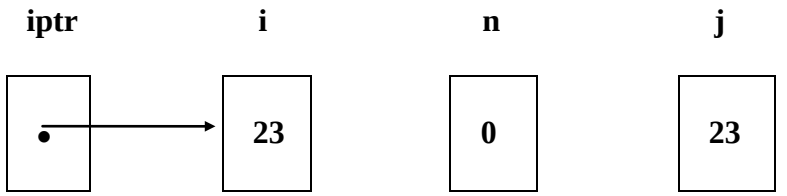
i = 0 ; iptr = &i ; n = 17 ; j = 23 ;

(**b**)



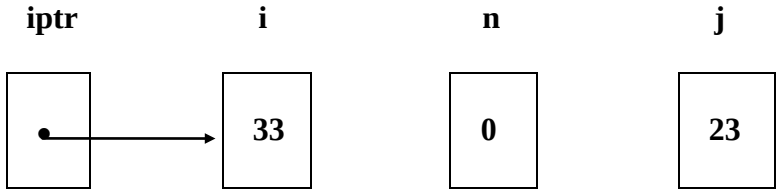
n = ∗iptr ;

(**c**)



```
*iptr = j ;
```

(**d**)




```
*iptr = *iptr + 10 ;
```

Because **iptr** is of type "pointer to **int**," we can use ***iptr** anywhere a variable can occur, such as in assignments. The assignments

```
n = *iptr ;
```

assigns to n whatever **iptr** points to, zero in this example. Similarly, the assignment

```
*iptr = j ;
```

assigns the j's current value to whatever iptr points to, i in this case. Similarly the assignment

```
*iptr = *iptr + 10 ;
```

adds 10 to whatever *iptr points to. Above figure illustrates what's going on with these assignments.

Pointer variables, like other C variables, are not automatically initialized. In fact a pointer variable starts off with a random value-whatever happens to be in the memory location reserved for the pointer-and could therefore point anywhere. That means that ***iptr** is meaningless until **iptr** has been made to point to something. Warning: **Don't dereference a pointer variable until you've assigned it an address**.

One way to prevent problems with using uninitialized pointer variables is to initialize all of them to **NULL**, a special pointer that points to nowhere. **NULL** is equivalent to the constant 0. In C, it is illegal to dereference 0, and in most environments doing so causes a run-time error that terminates the program. This makes accidentally dereferencing NULL cause less harm than dereferencing a random pointer.

NULL is defined in **stddef.h**, so any program that uses the NULL pointer should have the preprocessor directive

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

As an alternative, the program can simply use zero in place of NULL, or can precede its use with the statement

```
# define NULL  0
```

We will see additional uses of the null pointer throughout the course.

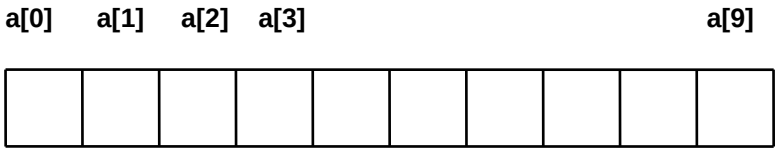
7.6 Pointers and Arrays

In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously. Any operation that can be achieved by array subscripting can also be done with pointers. The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand.

The declaration

```
int  a[10] ;
```

defines an array **a** of size 10, that is, a block of 10 consecutive objects named a[0], a[1], a[2], ... a[9].



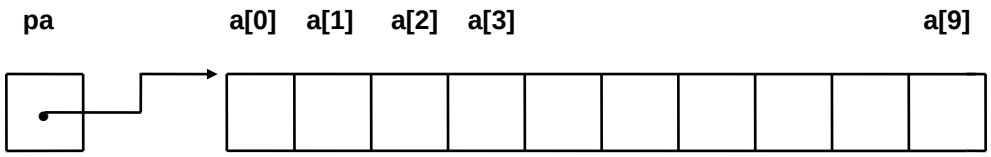
The notation a[i] refers to the i-th element of the array. If pa is a pointer to an integer, declared as

```
int  *pa ;
```

then the assignment

```
pa = &a[0] ;
```

sets pa to point to element zero of a; that is, pa contains the address of a[0].



Now the assignment

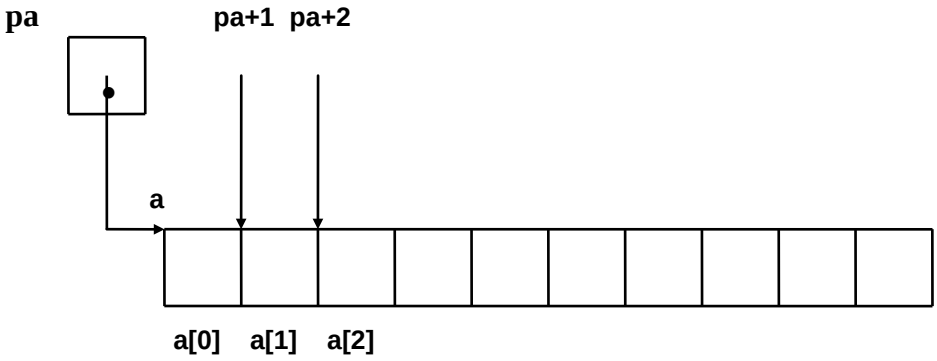
```
x = *pa ;
```

will copy the contents of a[0] into x.

If pa points to a particular element of an array, then by definition pa+1 points to the next element, pa+i points i elements after pa, and pa-i points i elements before. Thus, if pa points to a[0],

```
*( pa + 1 )
```

refers to the contents of a[1], pa+i is the address of a[i], and *(pa + i) is the contents of a[i].



These remarks are true regardless of the type or size of the variables in the array a. The meaning of "**adding 1 to a pointer**," and by extension, all pointer arithmetic, is that pa+1 points to the next object, and pa+i points to the i-th object beyond pa.

The correspondence between indexing and pointer arithmetic is very close. By definition, the value of a variable or expression of type array is the address of element zero of the array. Thus after the assignment

```
pa = &a[0] ;
```

pa and a have identical values. Since the name of an array is a synonym for the location of the initial element, the assignment

```
pa = &a[0] ;
```

can also be written as

```
pa = a ;
```

Rather more surprising, at least at first sight, is the fact that a reference to a[i] can also be written as *(a + i). In evaluating a[i], C converts it to *(a + i) immediately; the two forms are equivalent. Applying the operator & to both parts of this equivalence, it follows that &a[i] and a+i are also identical: a+i is the address of the i-th element beyond a. As the other side of this coin, if pa is a pointer, expressions may use it with a subscript; pa[i] is identical to *(pa + i). In short, an array-and-index expression is equivalent to one written as a pointer and offset.

There is one difference between an array name and a pointer that must be kept in mind. A pointer is a variable, so **pa=a** and **pa++** are legal. But an array name is not a variable; constructions like **a=pa** and **a++** are illegal.

When an array name is passed to a function, what is passed is the location of the initial element. Within the called function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address.

As formal parameters in a function definition,

```
char  S[ ] ;
```

and

```
char  *S ;
```

are equivalent; we prefer the latter because it says more explicitly that the parameter is a pointer. When an array name is passed to a function, the function can at its convenience believe that it has been handed either an array or pointer, and manipulate it accordingly. It can even use both notations if it seems appropriate and clear.

It is possible to pass part of an array to a function, by passing a pointer to the beginning of the subarray. For example, if a is an array,

```
f( &a[2] )
```

and

```
f( a+2 )
```

both pass to the function f the address of the subarray that starts at a[2]. Within f, the parameter declaration

```
f( int  arr[ ] )   {  ........  }
```

and

```
f( int  *arr )     {  ........  }
```

So as far as f is concerned, the fact that the parameter refers to part of a large array is of no consequence.

If one is sure that the elements exist, it is also possible to index backwords in an array; p[-1], p[-2], and so on are syntactically legal, and refer to the elements that immediately precede p[0]. Of course, it is illegal to refer to object that are not within the array bounds.

Eg : A Program that uses array subscripting to print the first "n" elements of an array.

```
# include <stdio.h>

# define  MAX  20

void print_table( int [ ] , int ) ;

void main(void)
{
        int table[MAX] ;
        int i ;

        for(  i = 0 ;  i < MAX ;  i++ )

                table[i] = i ;

        print_table( table , MAX ) ;

}

void print_table( int a[ ] , int n )
{
        int  i ;

        for( i = 0 ;  i < n ;  i++ )

                printf("%d\n" , a[i] ) ;
}
```

Eg : A Program that uses pointer indexing to print the first "n" elements of an array.

```
# include <stdio.h>

# define  MAX  20

void print_table( int [ ] , int ) ;

void main(void)
{
        int table[MAX] ;
        int i ;

        for(  i = 0 ;  i < MAX ;  i++ )

                table[i] = i ;

        print_table( table , MAX ) ;

}

void print_table( int a[ ] , int n )
{
        int  i ;
        int  *ptr ;

        for( ptr = a , i = 0  ;  i < n  ;  ptr++ , i++ )

                printf("%d\n" , *ptr ) ;

}
```

Eg : A Program that uses pointer indexing and comparison to print the first "n" elements of an array.

```
# include <stdio.h>

# define  MAX  20

void print_table( int [ ] , int ) ;



void main(void)
{
```

```
        int table[MAX] ;
        int i ;

        for(  i = 0 ;  i < MAX ;  i++ )

                table[i] = i ;

        print_table( table , MAX ) ;

}

void print_table( int a[ ] , int n )
{
        int  *ptr  ,  *endptr ;

        endptr = a + n - 1 ;

        for( ptr = a  ;  ptr <= endptr  ;  ptr++ )

                printf("%d\n" , *ptr ) ;

}
```

We can use pointer comparisons to write **print_table** more efficiently, and we have done so in above program. This loop is more efficient because it no longer tests a counter to determine when the array has been traversed. Instead, we compare the indexing pointer with the address of the array's last element.

In fact, we can write the loop to print table's values even more compactly.

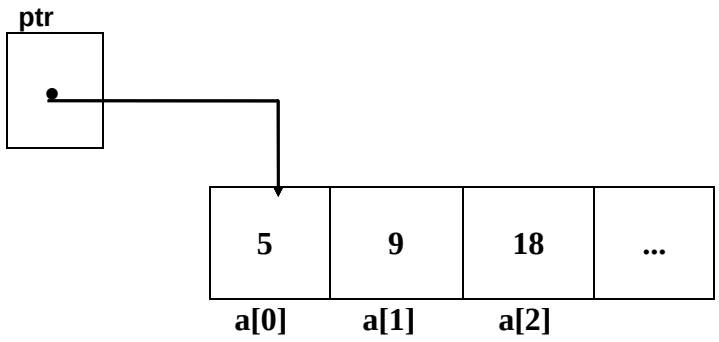
```
endptr  =  ( ptr = table )  + n  - 1 ;

while( ptr <= endptr )

        printf("%d\n" , * ptr++ ) ;
```

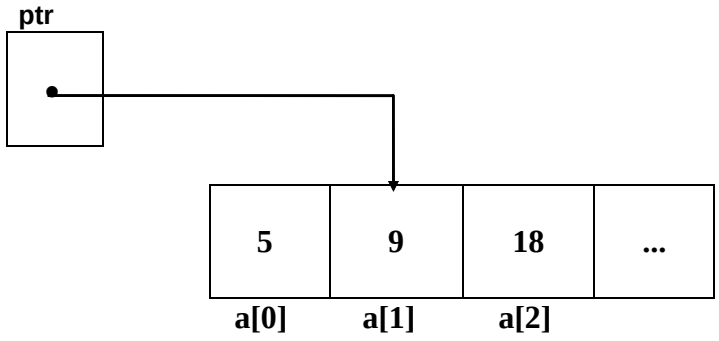
Because of the precedence and evaluation order of * and ++, *ptr++ means "**obtain the value that ptr points to (i.e. *ptr), return the value, and then increment the pointer**." This differs from (*ptr)++, which simply increments the value to which ptr points, after returning its original value. Similarly, *ptr-- decrements the pointer after returning the pointed-to value. The prefix forms increment (*++ptr) or (*--ptr) the pointer and return whatever value it then points to. Following figure illustrates how these expressions differ.

(**a**)



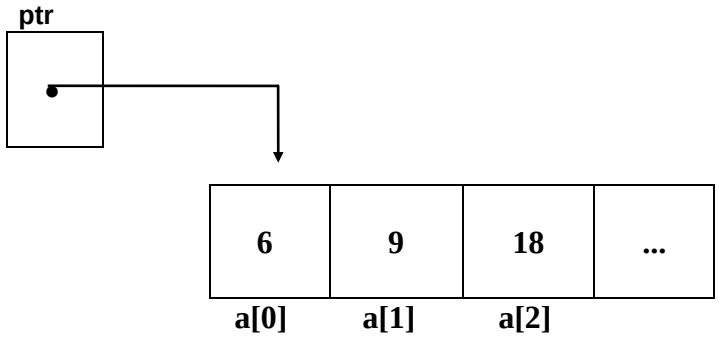
initially assume ptr = & a[0]

(**b**)



∗ptr++ increments ptr and returns 5

(**c**)



(∗ptr)++ returns 5 and increments a[0]

We saw earlier that when we pass an array parameter, we really only pass the address of its first element-or; in other words, a pointer. This means that

```
void print_table( int a[ ] , int n )
{

        ......

}
```

is equivalent to

```
void print_table( int  *a , int n )
{

        ......

}
```

When we call **print_table(table , MAX)**, the address of **table[0]** is copied into a. Because we only pass a pointer, we don't indicate bounds on table. Within **print_table**, we can access elements of a using either the array form **(a[i])** or the pointer form **(*(a + i))**. Pick your favourite, although it's customary to use the form corresponding to the way the object was defined.

Since C's parameters are passed by value (copied), we can use the passed pointer to traverse the array instead of declaring an additional local variable. We do this in one final version of **print_table**, shown below. When **print_table** is called, **ptr** is initialized with the address of the array's first element. We then use it to traverse the array, incrementing it after each element is printed. table is a constant and is not affected by the call to **print_table**.

As we do not explicitly declare an array parameter's size, functions that process arrays need to know how many array elements to process. That means that when we pass an array, we usually also pass the number of elements in the array, n in **print_table**. We cannot use **sizeof** in **print_table** because **sizeof(ptr)** returns the size of a pointer, not of the entire array, which is not necessarily known at compile time.

One nice benefit of an array being passed as a pointer is that we can pass the address of any array element. This effectively allows us to pass only part of an array. We can print k elements of table, starting with table[i], with either

```
print_table( &table[i] , k ) ;
```

or the equivalent

```
print_table( table + i , k ) ;
```

In both cases we pass **table[i]**'s address. The following figure shows how **print_table** can still access any element of table **through appropriate negative or positive offsets** of **ptr**.

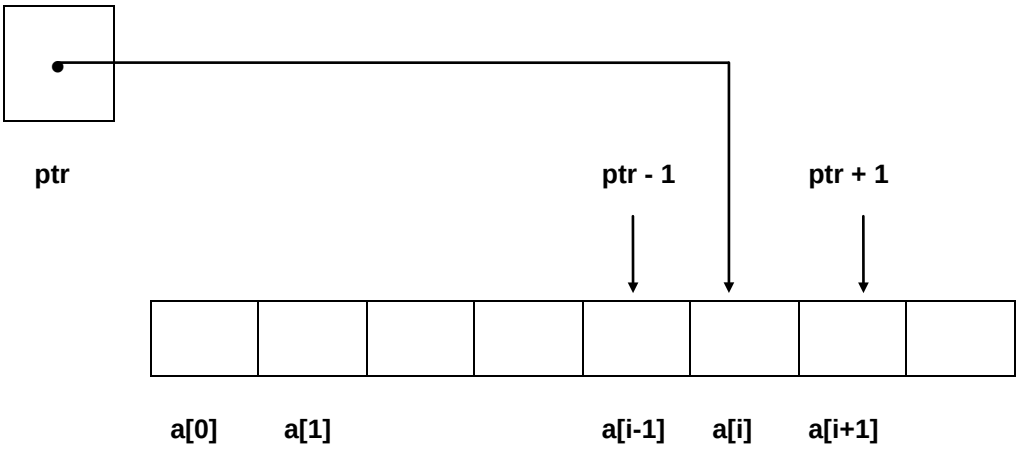


Figure : Passing part of an array. We can still access the entire array

Don't make the common mistake of passing an array element to a function expecting an array parameter. The call

```
print_table( table[i] , k ) ;          /* wrong : pointer not passed */
```

is a serious mistake, since **print_table** expects a pointer and instead receives an **int**. ***Warning : A function expecting an array must be passed an array name or an element's address***.

Eg : A Program that uses pointer indexing and comparison to print the first "n" elements of an array - most concise version.

```
# include <stdio.h>

# define  MAX  20

void print_table( int * ,  int  ) ;

void main(void)
{
        int table[MAX] ;
        int i ;

        for(  i = 0 ;  i < MAX ;  i++ )

                table[i] = i ;
        print_table( table , MAX ) ;

}
```

```
void print_table( int *ptr , int n )
{
        int  *endptr ;                          /* pointer to last element */

        for( endptr = ptr + n - 1  ;  ptr <= endptr  ;  ptr++ )

                printf("%d\n" , *ptr ) ;

}
```

7.7 Pointers to Pointers

In C, you can define pointer variables that point to other pointer variables, which, in turn, point to the data, such as a **char**. The following figure represents this relationship visually; char_ptr is a pointer variable that points to another pointer variable whose contents can be used to point to 'A'.

You may wonder why this is necessary. The advent of OS/2 and the Windows programming environment signals the development of multitasking operating environments designed to maximize the use of memory. In order to compact the use of memory, the operating system has to be able to move objects in memory whenever necessary. If your program points directly to the physical memory cell where the object is stored, and the operating system moves it, disaster will strike. Instead, your application points to a memory cell address that will not change while your program is running(a ***virtual_address***), and the ***virtual_address*** memory cell holds the ***current_physical_address*** of the data object. Now, whenever the operating environment wants to move the data object, the operating system just has to update the ***current_physical_address*** stored at the ***virtual_address***. As far as your application is concerned, it still uses the unchanged address
of the ***virtual_address*** to point to the updated address of the ***current_physical_address***.

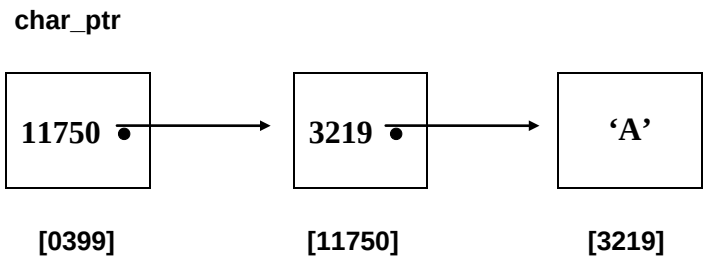


Figure : A pointer to a pointer of type char

To define a pointer to a pointer in C, you simply increase the number of asterisks preceding the identifier:

```
char  **char_ptr ;
```

In this example, the variable *char_ptr* is defined to be a pointer to a pointer that points to a **char** data type. *char_ptr's* data type is

```
char **
```

Each asterisk is read "**pointer to**." The number of pointers that must be followed to access the data item, or the number of asterisks that must be attached to the variable to reference the value to which it points, is called the ***level of indirection*** of the pointer variable. A pointer's level of indirection determines how much dereferencing must be done to access the data type given in the definition. Following figure illustrates several variables with different levels of indirecftion.

```
int  int_data  =  5 ;
int  *int_ptr1 ;
int  **int_ptr2 ;
int  ***int_ptr3 ;

int_ptr1 = &int_data ;
int_ptr2 = &int_ptr1 ;
int_ptr3 = &int_ptr2 ;
```

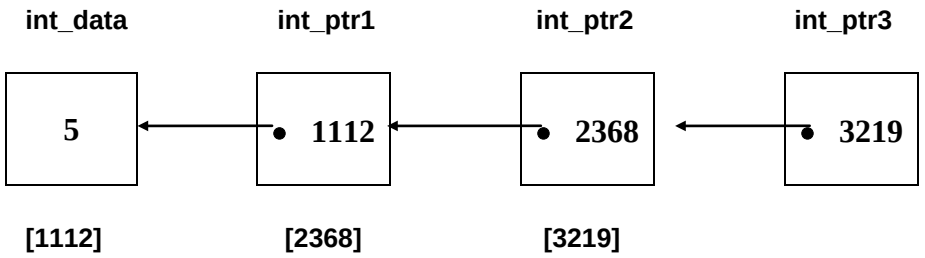


Figure : Three variables using different levels of indirection : *int_ptr1*, *int_ptr2*, and *int_ptr3*

The first four lines of code in above figure define three variables; the **int** variable int_data, the **int_ptr1** pointer variable that points to an **int** (one level of indirection), the **int_ptr2** variable that points to a pointer that points to an **int** (two level of indirection), and **int_ptr3**, which illustrates that this process can be extended beyond two levels of indirection. The fifth line of code

```
int_ptr1 = &int_data ;
```

is an assignment statement that uses the address operator. The expression assigns the address of &int_data to int_ptr1. Therefore, int_ptr1's contents include 1112. Notice that there is only one arrow from int_ptr1 to int_data. This indicates that int_data, or 5, can be accessed by dereferencing int_ptr1 just once. The next statement,

```
int_ptr2 = &int_ptr1 ;
```

along with its accompanying picture, illustrates double indirection. Because int_ptr2's data type is **int ****, to access an **int** you need to dereference the variable twice. After the preceding assignment statement, int_ptr2 holds the address of int_ptr1 (not the contents of int_ptr1); so int_ptr2 points to int_ptr1, which in turn points to int_data. Notice that you must follow two arrows to get from int_ptr2 to int_data.

The last statement demonstrates three levels of indirection

```
int_ptr3 = &int_ptr2 ;
```

and assigns the address of int_ptr2 to int_ptr3 (not the contents of int_ptr2). Note that the accompanying illustration shows that three arrows are now necessary to reference int_data.

To review, int_ptr3 is assigned the address of a pointer variable that indirectly points to an **int**, as in the previous statement. However, ***int_ptr3 (the cell pointed to) can only be assigned an **int** value, not an address.

```
***int_ptr3 = 10 ;
```

since ***int_ptr3 is an **int**.

C allows you to initialize pointers like any other variable. For example, you could have defined and initialized int_ptr3 with the following single statement :

```
int ***int_ptr3 = &int_ptr2 ;
```

7.8 Character Pointers and Functions

A string constant, written as

```
"I am a string"
```

is an array of characters. In the internal representation, the array is terminal with the null character '**\0**' so that programs can find the end. The length storage is thus one more than the number of characters between the double quotes.

Perhaps the most common occurrence of string constants is as argument in functions, as in

```
printf("Hello World\n") ;
```

When a character string like this appears in a program, access to it is through character pointer; **printf** receives a pointer to the beginning of the character array. That is, a string constant is accessed by a pointer to its first element.

String constants need not be function arguments. If p is declared as

```
char *p ;
```

then the statement

```
p = "now is the time" ;
```

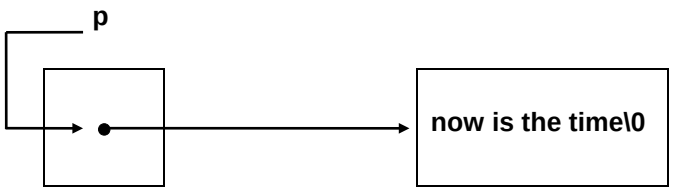
assigns to p a pointer to the character array. This is not a string copy; only pointers are involved. C does not provide any operators for providing an entire string of characters as a unit.

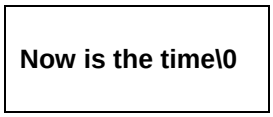
There is an important difference between these definitions :

```
char message[ ] = "now is the time" ;        /* an array */

char *p = "now is the time" ;                /* a pointer */
```

message is an array, just big enough to hold the sequence of characters upto '**\0**' that initializes it. Individual characters within the array may be char but message will always refer to the same storage. On the other hand p is a pointer, initialized to point to a string constant; the pointer subsequently be modified to point elsewhere, but the result is undefined if try to modify the string contents.





We will illustrates more aspects of pointers and arrays by studying useful functions adapted from the standard library.

Eg : A program to computes the length of a string - **strlen** function

(**a**) Using Arrays

```
# include <stdio.h>

int  strlen ( char [ ] ) ;

void main(void)
{
        char S[ ]  =  "University of Colombo" ;

        printf("%d\n", strlen( S ) ) ;
}

int  strlen ( char S[ ] )
{
        int  i = 0 , count = 0 ;

        while( S[i] != '\0' )
        {
                i++ ;
                count++ ;
        }

        return (count) ;
}
```

We can rewrite the above **strlen** function as

```
int  strlen ( char S[ ] )
{
        int  count = 0 ;

        while( S[count] != '\0' )

                count++ ;

        return (count) ;
}
```

or

```
int  strlen ( char S[ ] )
{
        int  count = 0 ;
```

```
        while( S[count] )

                count++ ;

        return (count) ;
}
```

Eg : A program to computes the length of a string - **strlen** function

(**b**) Using Pointers

```
# include <stdio.h>

int  strlen ( char  * ) ;

void main(void)
{
        char  *S  =  "University of Colombo" ;

        printf("%d\n", strlen( S ) ) ;
}

int  strlen ( char  *S )
{
        int  count = 0 ;

        while(  *S )
        {
                S++ ;
                count++ ;
        }

        return (count) ;
}
```

We can rewrite the above **strlen** function as

```
int  strlen ( char  *S )
{
        int  count ;

        for( count = 0   ;  *S   ;  S++  )

                count++ ;
```

```
        return (count) ;
}
```

or

```
int  strlen ( char  *S )
{
        char  *P  =  S ;

        while(  *P )

                P++ ;

        return ( P - S ) ;
}
```

Eg : A program to concatenate a string to end of another string - **strcat** function

(**a**) Using Arrays

```
# include <stdio.h>

char  strcat (  char [ ]  ,  char [ ]  ) ;

void main(void)
{
        char S1[ ]  =  "University of Colombo" ;

        char S2[ ]  =  "Sri Lanka" ;

        printf("%s\n", strcat(  S1 , S2  ) ) ;
}

char  strcat (  char S1[ ]  ,  char S2[ ]  )
{
        int  i = 0 , j = 0 ;

        while(  S1[i]  )                        /*  To find end of S1  */

                i++ ;

        while(  S1[i] = S2[j] )                 /*  To copy S2 into S1  */
        {
                i++ ;
                j++ ;
```

```
        }

        return ( S1 ) ;
}
```

Eg : A program to concatenate a string to end of another string - **strcat** function

(**b**) Using Pointers

```
# include <stdio.h>

char *strcat ( char * , char * ) ;

void main(void)
{
        char *S1 = "University of Colombo" ;

        char *S2 = "Sri Lanka" ;

        printf("%s\n", strcat( S1 , S2 ) ) ;
}

char *strcat ( char *S1 , char *S2 )
{
        char *P = S1 ;

        while( *S1 )                    /* To find end of S1 */

                S1++ ;

        while( *S1 = *S2 )              /* To copy S2 into S1 */
        {
                S1++ ;
                S2++ ;
        }

        return ( P ) ;
}
```

Eg : A program to reverse a string - **strrev** function

(**a**) Using Arrays

```
# include <stdio.h>
```

```
char  strrev ( char [ ] ) ;

void main(void)
{
        char S[ ] = "University of Colombo" ;

        printf("%s\n", strrev( S ) ) ;
}

char  strrev ( char S[ ] )
{
        int i , j , C ;

        for( i = 0 , j = strlen( S ) - 1 ; i < j ; i++ , j++ )
        {
                C = S[i] ;

                S[i] = S[j] ;

                S[j] = C ;
        }

        return ( S ) ;
}
```

Eg : A program to reverse a string - **strrev** function

(**b**) Using Pointers

```
# include <stdio.h>

char *strrev ( char * ) ;

void main(void)
{
        char *S = "University of Colombo" ;

        printf("%s\n", strrev( S ) ) ;
}

char *strrev ( char *S )
{
        char *P , *Q , temp ;
```

```
        for( P = S , Q = P + strlen( S ) - 1 ; P < Q ; P++ , Q-- )
        {
                temp = *P ;

                *P = *Q ;

                *Q = temp ;
        }

        return ( S ) ;
}
```

We can rewrite the above **strrev** function as

```
char *strrev ( char *S )
{
        char *P , *Q , temp ;

        int n = strlen( S ) ;

        Q = ( n > 0 ) ? S + n - 1 : S ;

        for( P = S ; P < Q ; P++ , Q-- )
        {
                temp = *P ;

                *P = *Q ;

                *Q = temp ;
        }

        return ( S ) ;
}
```

Eg : A program to copy a string - **strcpy** function

(**a**) Using Arrays

```
# include <stdio.h>

# define MAX 50

char strcpy ( char [ ] , char [ ] ) ;
```

```
void main(void)
{
        char S1[ ] = "University of Colombo" ;

        char S2[MAX] ;

        printf("%s\n", strcpy( S1 , S2 ) ) ;
}

char strcpy ( char S1[ ] , char S2[ ] )
{
        int i = 0 ;

        while( S2[i] = S1[i] )

                i++ ;

        return ( S2 ) ;
}
```

Eg : A program to copy a string - **strcpy** function

(**b**) Using Pointers

```
# include <stdio.h>

char *strcpy ( char * , char * ) ;

void main(void)
{
        char *S1 = "University of Colombo" ;

        char *S2 ;

        printf("%s\n", strcpy( S1 , S2 ) ) ;
}

char *strcpy ( char *S1 , char *S2 )
{
        char *P = S2 ;

        while( *S2 = *S1 )
        {
                S1++ ;
```

```
            S2++ ;
        }

        return ( P ) ;
}
```

Eg : A program to compare two strings - **strcmp** function (version (i))

```
# include <stdio.h>

int strcmp ( char [ ] , char [ ] ) ;

void main(void)
{
        char S1[ ] = "University of Colombo" ;

        char S2[ ] = "Sri Lanka" ;

        printf("%d\n", strcmp( S1 , S2 ) ) ;
}

int strcmp ( char S1[ ] , char S2[ ] )
{
        int answer , i = 0 ;

        while( S1[i] != '\0' && S2[i] != '\0' )

                i++ ;

        if( S1[i] == '\0' && S2[i] == '\0' )

                answer = 1 ;                    /* strings are equal */

        else

                answer = 0 ;                    /* not equal */

        return ( answer ) ;
}
```

Eg : A program to compare two strings - **strcmp** function (version (ii))

```
# include <stdio.h>
```

```
int  strcmp ( char [ ] , char [ ] ) ;

void main(void)
{
        char S1[ ] = "University of Colombo" ;

        char S2[ ] = "Sri Lanka" ;

        printf("%d\n", strcmp( S1 , S2 ) ) ;
}

int  strcmp ( char S1[ ] , char S2[ ] )
{
        int answer , i = 0 ;

        while( S1[i] != '\0' && S2[i] != '\0' )

                i++ ;

        if( S1[i] < S2[i] )                     /* S1 < S2 */

                answer = -1 ;

        else if ( S1[i] == S2[i] )              /* S1 = S2 */

                answer = 0 ;

        else                                    /* S1 > S2 */

                answer = 1 ;

        return ( answer ) ;
}
```

Eg : A program to compare two strings - **strcmp** function (version (iii))

```
# include <stdio.h>

int  strcmp ( char [ ] , char [ ] ) ;

void main(void)
{
        char S1[ ] = "University of Colombo" ;

        char S2[ ] = "Sri Lanka" ;
```

```
        printf("%d\n", strcmp( S1 , S2 ) ) ;
}

int strcmp ( char S1[ ] , char S2[ ] )
{
        int answer , i = 0 ;

        while( S1[i] != '\0' && S2[i] != '\0' )

                i++ ;

        if( ( S1[i] =='\0' ) && ( S2[i] != '\0' ) )          /* S1 < S2 */

                answer = -1 ;

        if( ( S1[i] == '\0' ) && ( S2[i] == '\0' ) )         /* S1 = S2 */

                answer = 0 ;

        if( ( S1[i] != '\0' ) && ( S2[i] == '\0' ) )         /* S1 > S2 */

                answer = 1 ;

        return ( answer ) ;
}
```

Eg : A program to compare two strings - **strcmp** function

(**a**) Using Arrays

```
# include <stdio.h>

int strcmp ( char [ ] , char [ ] ) ;

void main(void)
{
        char S1[ ] = "University of Colombo" ;

        char S2[ ] = "Sri Lanka" ;

        printf("%d\n", strcmp( S1 , S2 ) ) ;
}

int strcmp ( char S1[ ] , char S2[ ] )
```

```
{
        int  i  =  0 ;

        while(  S1[i]  && S2[i]  )

                i++ ;

        return ( S1[i] - S2[i]  ) ;
}
```

Eg : A program to compare two strings - **strcmp** function

(**a**) Using Pointers

```
# include <stdio.h>

int  strcmp ( char * , char * ) ;

void main(void)
{
        char  *S1  =  "University of Colombo" ;

        char  *S2  =  "Sri Lanka" ;

        printf("%d\n", strcmp( S1 , S2 ) ) ;
}

int  strcmp ( char  *S1 , char  *S2 )
{
        while(  *S1 && *S2 )
        {

                S1++ ;

                S2++ ;
        }

        return ( *S1- *S2 ) ;
}
```