UNIT 7: POINTERS AND MEMORY ALLOCATION

UNIT STRUCTURE

7.0 Objectives
7.1 Introduction to Ponters
7.2 Pointer Operators
  7.2.1 Declaration of pointer
  7.2.2 Reference operator and dereference operator
  7.2.3 Null Pointers
  7.2.4 Pointer to Pointer
7.3 Pointer Arithmetic
7.4 Pointers and Arrays
7.5 Array of Pointers
7.6 Pointers and Strings
7.7 Pointer Indirection
  7.7.1 One Dimensional Array and Pointer Indirection Operator
  7.7.2 Multi Dimensional Array and Pointer Indirection Operator
7.8 Pointers to Functions
7.9 Dynamic Memory Allocation
  7.9.1 malloc() function
  7.9.2 calloc() function
  7.9.3 realloc() function
  7.9.4 free() function
  7.9.5 Difference between Static memory and dynamic memory
  7.9.6 difference between malloc() and calloc()
7.10 Summary
7.11 Questions for Exercises
7.12 Suggested Readings

7.0 Objectives

After going through this unit you should be able to :

- understand the concept of pointers
- address and use od indirection operators
- use the pointer arithmetic
- handle pointers to functions
- understand the concept of dynamic memory allocation

7.1 INTRODUCTION TO POINTERS

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps.

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is −

*type *var-name;*

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations −

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory.

If you have a variable *var* in your program, *&var* will give you its address in the memory, where *&* is commonly called the *reference operator*.
You must have seen this notation while using scanf() function. It was used in the function to store the user inputted value in the address of var.

```
scanf("%d", &var);
```

Example 7.1

```
/* Example to demonstrate use of reference operator in C programming. */
#include <stdio.h>
int main()
{
  int var = 5;
  printf("Value: %d\n", var);
  printf("Address: %u", &var);  //Notice, the ampersand(&) before var.
  return 0;
}
```

Output
Value: 5
Address: 2686778

7.2 Pointer Operators

In C, there is a special variable that stores just the address of another variable. It is called Pointer variable or, simply, a pointer.

7.2.1 Declaration of Pointer

data_type pointer_variable_name;*

For example,
```
int* p;
```

Above statement defines, p as pointer variable of type int.

7.2.2 Reference operator (&) and Dereference operator (*)

As discussed, & is called reference operator. It gives you the address of a variable.
Likewise, there is another operator that gets you the value from the address, it is called a dereference operator (*).

There are a few important operations, which we will do with the help of pointers very frequently.
(a) We define a pointer variable,
(b) assign the address of a variable to a pointer and
(c) finally access the value at the address available in the pointer variable.

Below example clearly demonstrates the use of pointers, reference operator and dereference operator.

Note: The * sign when declaring a pointer is not a dereference operator. It is just a similar notation that creates a pointer.

Example 7.2 :

```
/* Source code to demonstrate, handling of pointers in C program */
#include <stdio.h>
int main()
{
   int* pc;
   int c;
   c=22;
   printf("Address of c:%u\n",&c);
   printf("Value of c:%d\n\n",c);
   pc=&c;
   printf("Address of pointer pc:%u\n",pc);
   printf("Content of pointer pc:%d\n\n",*pc);
   c=11;
   printf("Address of pointer pc:%u\n",pc);
   printf("Content of pointer pc:%d\n\n",*pc);
   *pc=2;
   printf("Address of c:%u\n",&c);
   printf("Value of c:%d\n\n",c);
   return 0;
}
```

Output

```
Address of c: 2686784
Value of c: 22

Address of pointer pc: 2686784
Content of pointer pc: 22

Address of pointer pc: 2686784
Content of pointer pc: 11

Address of c: 2686784
Value of c: 2
```

7.2.3 NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.
The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program –

```
#include <stdio.h>
int main ()
 {
   int  *ptr = NULL;
   printf("The value of ptr is : %x\n", ptr  );
    return 0;
}
```

Output:

When the above code is compiled and executed, it produces the following result −

The value of ptr is 0

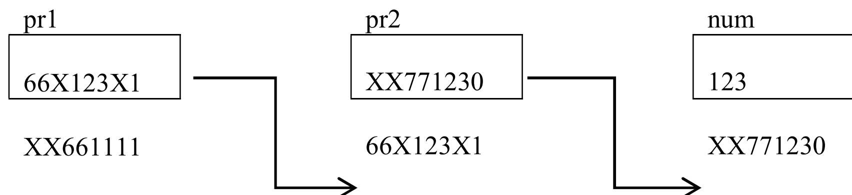
In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows –

```
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

7.2.4 Pointer to Pointer

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following declaration declares a pointer to a pointer of type int –

*int **var;*

Example 7.2.4 : To demonstrate pointer to a pointer

```
#include <stdio.h>
int main()
{
    int num=123;
    int *pr2;          //Pointer for num
    int **pr1;         //Double pointer for pr2
    pr2 = & num;      //  the address of variable num is stored  in pointer pr2
     pr1 = &pr2;     // storing the address of pointer pr2 into another pointer pr1

    /* Possible ways to find value of variable num*/
    printf("\n Value of num is: %d", num);
    printf("\n Value of num using pr2 is: %d", *pr2);
    printf("\n Value of num using pr1 is: %d", **pr1);

    /*Possible ways to find address of num*/
    printf("\n Address of num is: %u", &num);
    printf("\n Address of num using pr2 is: %u", pr2);
    printf("\n Address of num using pr1 is: %u", *pr1);

    /*Find value of pointer*/
    printf("\n Value of Pointer pr2 is: %u", pr2);
    printf("\n Value of Pointer pr2 using pr1 is: %u", *pr1);
```

```
    /*Ways to find address of pointer*/
    printf("\n Address of Pointer pr2 is:%u",&pr2);
    printf("\n Address of Pointer pr2 using pr1 is:%u",*pr1);

    /*Double pointer value and address*/
    printf("\n Value of Pointer pr1 is:%u",pr1);
    printf("\n Address of Pointer pr1 is:%u",&pr1);

    return 0;
}
```

Output:

Value of num is: 123
Value of num using pr2 is: 123
Value of num using pr1 is: 123
Address of num is: XX771230
Address of num using pr2 is: XX771230
Address of num using pr1 is: XX771230
Value of Pointer pr2 is: XX771230
Value of Pointer pr2 using pr1 is: XX771230
Address of Pointer pr2 is: 66X123X1
Address of Pointer pr2 using pr1 is: 66X123X1
Value of Pointer pr1 is: 66X123X1
Address of Pointer pr1 is: XX661111

Below points will help out to clarify few things in a better way –

num = *pr2 = **pr1
&num = pr2 = *pr1
&pr2 = pr1

7.3 POINTER ARITHMETIC

A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, let us consider that ptr is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer:

```
ptr++
```

After the above operation, the ptr will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to the next memory location without impacting the actual value at the memory location. If ptr points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character will be available at 1001.
The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type. Thus, the following operations can be performed on a pointer:

(a) Addition of a number to a pointer. For example,

```
int i = 4, *j, *k ;
j = &i ;
j = j + 1 ;
j = j + 9 ;
k = j + 3 ;
```

(b) Subtraction of a number from a pointer. For example,

```
j = &i ;
```

```
j = j - 2 ;
j = j - 5 ;
k = j - 6 ;
```

(c) Subtraction of one pointer from another

One pointer variable can be subtracted from another provided both variables point to elements of the same array. The resulting value indicates the number of bytes separating the corresponding array elements. This is illustrated in the following program.

```
main( )
{
int arr[ ] = { 10, 20, 30, 45, 67, 56, 74 } ;
int *i, *j ;
i = &arr[1] ;
j = &arr[5] ;
printf ( "%d %d", j - i, *j - *i ) ;
}
```

Here **i** and **j** have been declared as integer pointers holding addresses of first and fifth element of the array respectively. Suppose the array begins at location 65502, then the elements **arr[1]** and **arr[5]** would be present at locations 65504 and 65512 respectively, since each integer in the array occupies two bytes in memory. The expression **j - i** would print a value 4 and not 8. This is because **j** and **i** are pointing to locations that are 4 integers apart.

(d) Comparison of two pointer variables

Pointer variables can be compared provided both variables point to objects of the same data type. Such comparisons can be useful when both pointer variables point to elements of the same array. The comparison can test for either equality or inequality. Moreover, a pointer variable can be compared with zero (usually expressed as NULL). The following program illustrates how the comparison is carried out.

```
main( )
{
int arr[ ] = { 10, 20, 36, 72, 45, 36 } ;
int *j, *k ;
j = &arr [ 4 ] ;
k = ( arr + 4 ) ;
if ( j == k )
printf ( "The two pointers point to the same location" ) ;
else
printf ( "The two pointers do not point to the same location" ) ;
}
```

7.4 POINTERS AND ARRAYS

Let us consider a block in memory consisting if ten integers in a row. That is, 20 bytes of memory are set aside to hold 10 integers. Now, let's say we point our integer pointer ptr at the first of these integers. Furthermore lets say that integer is located at memory location 100 (decimal). What happens when we write: ptr + 1; Because the compiler "knows" this is a pointer (i.e. its value is an address) and that it points to an integer (its current address, 100, is the address of an integer), it adds 2 to ptr instead of 1, so the pointer "points to" the next integer, at memory location 102.

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. When performing addition on a pointer, size of a pointer increments by no. of storage units and not by no. of bytes. This is the beauty of scaling that pointer arithmetic doesn’t depend on type pointer is pointing to. Meaning that if a pointer points to character, when incremented by 1, points to next character, or if it points to float, points to next float when incremented by 1 and so on.

Example 7.4 : The following program increments the variable pointer to access each succeeding element of the array –

```
#include <stdio.h>
const int MAX = 3;
int main ()
 {
   int  var[] = {10, 100, 200};
   int  i, *ptr;
   /* let us have array address in pointer */
   ptr = var;

   for ( i = 0; i < MAX; i++)
{
      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %d\n", i, *ptr );
      /* move to the next location */
      ptr++;
   }

   return 0;
}
```

Output:

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

Accessing array elements by pointers is **always** faster than accessing them by subscripts. However, from the point of view of convenience in programming we should observe the following:
Array elements should be accessed using pointers if the elements are to be accessed in a fixed order, say from beginning to end, or from end to beginning, or every alternate element or any such
definite logic. Instead, it would be easier to access the elements using a subscript if there is no fixed logic in accessing the elements. However, in this case also, accessing the elements by pointers would work faster than subscripts.

7.5 ARRAY OF POINTERS

There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer

```
int *ptr[MAX];
```

It declares ptr as an array of MAX integer pointers. Thus, each element in ptr, holds a pointer to an int value. The following example uses three integers, which are stored in an array of pointers, as follows
Example 7.5.1

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

const int MAX = 3;

int main () {

   int  var[] = {10, 100, 200};
   int i, *ptr[MAX];
```

```
  for ( i = 0; i < MAX; i++) {
    ptr[i] = &var[i]; /* assign the address of integer. */
  }

  for ( i = 0; i < MAX; i++) {
    printf("Value of var[%d] = %d\n", i, *ptr[i] );
  }

  return 0;
}
```

Example 7.5.2
Write a program in which two-dimensional array is represented as an array of integer pointers to a set of single-dimension integer arrays

```
# include <stdio.h>
#include <stdlib.h>
#define MAXROWS 3
void main()
{
        int *ptr1[MAXROWS], *ptr2[MAXROWS], *ptr3[MAXROWS];
        int rows, cols, i, j;
        void inputmat(int *[ ], int, int);
        void dispmat(int *[ ], int, int);
        void calcdiff (int *[ ], int *[ ], int, int);

        printf("Enter no. of rows & columns \n");
        scanf("%d%d, &rows, &cols);

        for (i=0;i<rows;i++)
        {
                ptr1[i]=(int *)malloc (cols * sizeof (int));
                ptr2[i]=(int *)malloc (cols * sizeof (int));
                ptr3[i]=(int *)malloc (cols * sizeof (int));
        }
        printf ("Enter values in first matrix \n");
        inputmat(ptr1, rows, cols);
        printf ("Enter values in second matrix \n");
        inputmat(ptr2, rows, cols);
        calcdiff(ptr1, ptr2, ptr3, rows,cols);
        printf("Display difference of the two matrices \n");
        dispmat(ptr3, rows, cols);
}

void inputmat(int *ptr1[MAXROWS], int m, int n)
{
        int i, j;
        for (i=0; i<m; i++)
        {
                for (j=0; j<n; j++)
                {
                        scanf("%d", (*(ptr1 + i) + j));
                }
        }
        return;
}

void dispmat ( int *ptr3[ MAXROWS], int m, int n)
```

```
{
	int i, j;
	for (i=0; i<m; i++)
	{
		for (j=0; j<n; j++)
		{
			printf("%d", *(*ptr3+i)+j));
		}
		printf("\n");
	}
	return;
}

void calcdiff(int *ptr1[MAXROWS], int *ptr2[MAXROWS], int *ptr3[MAXROWS], int m, int n)
{
	int i, j;
	for (i=0; i<m; i++)
	{
		for (j=0; j<n; j++)
		{
			*(*(ptr3+i)+j)= *(*(ptr1+i)+j) - *(*(ptr2+i)+j);
		}
	}
	return;
}
```

OUTPUT

```
Enter no. of rows & columns
3  3
Enter values in first matrix
5	4	3
7	8	9
3	2	1
Enter values in second  matrix
2	4	1
4	6	5
1	1	0
Display difference of the two matrices
3	0	2
3	2	4
2	1	1
```

In this program, *ptr1, ptr2, p*tr3 are each defined as an array of pointers to integers. Each array has a maximum of MAXROWS elements. Since each element of ptr1, ptr2, ptr3 is a pointer, we must provide each pointer with enough memory for each row of integers. This can be done using the library function *malloc* included in *stdlib.h* header file as follows:

ptr1[1] = (int)malloc(cols*sizeof(int));*

This function reserves a block of memory whose size(in bytes) is equivalent to *cols * sizeof(int).* Since cols=3, so 3*2(size of int data type)i.e., 6 is allocated to each ptr1[1], ptr1[2] and ptr1[3]. This malloc function returns a pointer of type *void*. This means that we can assign it to any type of pointer. In this case, the pointer is type-casted to an integer type and assigned to the pointer ptr1[1], ptr1[2] and ptr1[3] points to the first byte of the memory allocated to the corresponding set of one-dimensional integer arrays of the original two-dimensional array.

The process of calculating and allocating memory at run time is known as *dynamic memory allocation.* The library routine *malloc* can be used for this purpose.

Instead of using conventional array notation, pointer notation has been used for accessing the address and value of corresponding array elements.the difference of the array elements within the function *calcdiff* is written as
((ptr3 + i) +j) = *(*(ptr1 + i) +j) - *(*(ptr2 + i) +j)

7.6 POINTERS AND STRINGS

Suppose we wish to store "Hello". We may either store it in a string or we may ask the C compiler to store it at some location in memory and assign the address of the string in a **char** pointer. This
is shown below:

```
char str[ ] = "Hello" ;
char *p = "Hello" ;
```

There is a subtle difference in usage of these two forms. For example, we cannot assign a string to another, whereas, we can assign a **char** pointer to another **char** pointer. This is shown in the
following program.

```
main( )
{
        char str1[ ] = "Hello" ;
        char str2[10] ;
        char *s = "Good Morning" ;
        char *q ;
        str2 = str1 ; /* error */
        q = s ; /* works */
}
```

Also, once a string has been defined it cannot be initialized to another set of characters. Unlike strings, such an operation is perfectly valid with **char** pointers.

```
main( )
{
        char str1[ ] = "Hello" ;
        char *p = "Hello" ;
        str1 = "Bye" ; /* error */
        p = "Bye" ; /* works */
}
```

Example 7.6.1

Program to reverse a string and check whether it is a palindrome using pointers

```
#include <stdio.h>

int is_palindrome(char*);
void copy_string(char*, char*);
void reverse_string(char*);
int string_length(char*);
int compare_string(char*, char*);

int main()
{
  char string[100];
  int result;

  printf("Input a string\n");
  gets(string);
```

```
  result = is_palindrome(string);

  if ( result == 1 )
    printf("\"%s\" is a palindrome string.\n", string);
  else
    printf("\"%s\" is not a palindrome string.\n", string);

  return 0;
}

int is_palindrome(char *string)
{
  int check, length;
  char *reverse;

  length = string_length(string);
  reverse = (char*)malloc(length+1);

  copy_string(reverse, string);
  reverse_string(reverse);

  check = compare_string(string, reverse);

  free(reverse);

  if ( check == 0 )
    return 1;
  else
    return 0;
}

int string_length(char *string)
{
  int length = 0;

  while(*string)
  {
    length++;
    string++;
  }

  return length;
}

void copy_string(char *target, char *source)
{
  while(*source)
  {
    *target = *source;
    source++;
    target++;
  }
  *target = '\0';
}

void reverse_string(char *string)
{
  int length, c;
```

```
  char *begin, *end, temp;

  length = string_length(string);

  begin = string;
  end = string;

  for ( c = 0 ; c < ( length - 1 ) ; c++ )
     end++;

  for ( c = 0 ; c < length/2 ; c++ )
  {
    temp = *end;
    *end = *begin;
    *begin = temp;

    begin++;
    end--;
  }
}

int compare_string(char *first, char *second)
{
  while(*first==*second)
  {
    if ( *first == '\0' || *second == '\0' )
      break;

    first++;
    second++;
  }
  if( *first == '\0' && *second == '\0' )
    return 0;
  else
    return -1;
}
```

7.7 POINTER INDIRECTION

An indirection operator is an operator used to obtain the value of a variable to which a pointer points. While a pointer pointing to a variable provides an indirect access to the value of the variable stored in its memory address, the indirection operator dereferences the pointer and returns the value of the variable at that memory location. The indirection operator is a unary operator represented by the symbol (*).

The indirection operator can be used in a pointer to a pointer to an integer, a single-dimensional array of pointers to integers, a pointer to a char, and a pointer to an unknown type.

The indirection operator is also known as the dereference operator.

A pointer variable is like an ordinary variable in that it is allocated location, by compiler, in memory when declared in the program. But unlike ordinary variable, a pointer holds address. Since memory is Byte addressable, every byte has address. Address itself is a constant value. Pointer holds addresses, meaning that what address does it hold, it points to that location in memory.

Example 7.7.1:

```
/* Program shows how pointer is assigned an address */
#include <stdio.h>
```

```
int main(void)
{
   int i = 100;
   int *ip = &i;
   /* ip is declared and initialized with address of integer i */

   printf("Address of i is %p \nand Value of ip"
         " in exp. \"*ip = &i\" is %p\n", &i, ip);

   return 0;
}
```

Output of the above program is as follows:

Address of i is 0x7fffb6801824
and Value of ip in exp. "*ip = &i" is 0x7fffb6801824

Observe here that both addresses are same. Value of pointer variable, which is address of integer i, points to the location allocated to integer i. But how to access the value at that location. In simple ways, integer ‘i’ gives value at that location.

Example 7.7.2:
/ Program shows how pointer is dereferenced or performed upon indirection */*

```
#include <stdio.h>

int main(void)
{
   int i = 100;
   int *ip = &i;
   /* "ip" is declared and initialized with address of integer i */

   printf("\nAddress of i is %p\nand Value of "
         "ip in exp. \"*ip = &i\" is %p\n\n", &i, ip);

   printf("Value of integer i  is %d\n", i);
   printf("\nAccessing the value of i indirectly using\ninteger "
         "pointer ip in the exp. \"*ip = &i\" is %d\n\n", *ip);

   return 0;
}
```

Output of the above program is given below:
Address of i is 0x7ffffb7f6ce4
and Value of ip in exp. "*ip = &i" is 0x7ffffb7f6ce4

Value of integer i is 100

Accessing the value of integer i indirectly using
integer pointer ip in the exp. "*ip = &i" is 100

Notice in the last printf() statement, we accessed the value of the integer i indirectly using integer pointer ip. Since we have indirectly accessed i, we perhaps call this Indirection or Dereferencing the pointer and unary operator (*) the Indirection operator.

Pointer doesn’t have built-in property of Indirection. Preceding a pointer with unary (*) operator, like *ip; in the exp. *ip = &i, is read as “go to the location pointed to by pointer, and access the I-value at that location”.

Also note that Indirection operator is one of very few operators which makes pointer expressions as ‘modifiable values’ meaning that such expressions represent locations in memory and can be used on left side of assignment ‘=’ operator.

7.7.1 One Dimensional Array and Pointer Indirection Operator

If we have the declarations

```
int a, *b, c[], *(*d[])();
```

then, in the code, the expressions

a, *b, c[] and *(*d[])()

would all evaluate to an integer. Encountering the declaration, you might have a hard time figuring out that d is an array of pointers to functions which return integer pointers, but you do know what type it will evaluate to when used in the context given. Thus you know that the statement

```
a = *(*d[5])(x, y)
```

will place an integer in a, even if you are not sure what happened. You could similarly match types by stripping off matching levels of indirections:

```
b = (*d[5])(x, y)
```

would store an integer pointer in b rather than the value of the integer.
Although the expression given in the declaration is generally the correct way to use the variable, the relation between pointers and arrays allows for other uses. Since an array name is just a pointer, we can actually use the expressions

b[] and *c

as well. (Using the alternate notation is often confusing but occasionally more clear.) For example, the sixth element of an array, declared by either of the two methods mentioned above, can be accessed in either of the two following methods:

b[5] or *(b+5)

(Recall that the monadic "*" operator merely takes the value at the right and performs one level of indirection on it.) The second method adds 5*(size of the array element type) to the address of array, resulting in a pointer to the sixth element, and then the "*" causes an indirection on that address, resulting in the value stored there. The subscripted notation is merely shorthand for this.

7.7.2 Multi-dimensional arrays and Pointer Indirection Operator

C uses two implementations of arrays, depending on the declaration. They are the same for one dimension, but different for more dimensions.

For example, if an array is declared as

```
int array[10][20][30];
```

than there are exactly 6000 ints of storage allocated, and a reference of the form array[i][j][k]

will be translated to *(array + i*20*30 + j*30 + k) which calculates the correct offset from the pointer "array", and then does an indirection on it.

Here each dimension is represented by a vector of pointers of objects of the next dimension, except the last dimension, which consists of arrays of data.

If a three-dimensional array is declared as

```
int ***array;
```

(and we will assume for the moment that it has been allocated space for a 10*20*30 array), then there is an array of 10 pointers to pointers to ints, 10 arrays of 20 pointers to ints, and 6000 ints. The 200 elements of the 10 arrays each point to a block of 30 ints, and the 10 elements of the one array each point to one of the 10 arrays. The array variable points to the head of the array with 10 elements.
In short, "array" points to a pointer to a pointer to an integer, "*array" points to a pointer to an integer, "**array" points to an integer, and "***array" is an integer.

In this case, an access of the form

```
array[i][j][k]
```

results in an access of the form

```
*( *( *(var+i) + j ) + k )
```

Which means: Take a pointer to the main array, add i to offset to the pointer to the correct second dimension array and indirect to it. Now we have a pointer to one of the arrays of 20 pointers, and we add j to get the offset to the next dimension, and we do an indirection on that. We now have a pointer to an array of 30 integers, so we add k to get a pointer to the desired integer, do an indirection, and we have the integer.

7.8 POINTERS TO FUNCTIONS

In C, like normal data pointers (int *, char *, etc), we can have pointers to functions. Following is a simple example that shows declaration and function call using function pointer.

Example 7.8.1

```
#include <stdio.h>

void fun(int a)       // A normal function with an int parameter  and void return type
{
   printf("Value of a is %d\n", a);
}

int main()
{
   void (*fun_ptr)(int) = &fun;        // fun_ptr is a pointer to function fun()
   /* The above line is equivalent of following two
      void (*fun_ptr)(int);
      fun_ptr = &fun;
   */

   (*fun_ptr)(10);           // Invoking fun() using fun_ptr
    return 0;
}
```

Output:

```
Value of a is 10
```

Why do we need an extra bracket around function pointers like fun_ptr in above example?
If we remove bracket, then the expression "void (*fun_ptr)(int)" becomes "void *fun_ptr(int)" which is declaration of a function that returns void pointer

Following are some interesting facts about function pointers.

1) Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.

2) Unlike normal pointers, we do not allocate de-allocate memory using function pointers.

3) A function's name can also be used to get functions' address. For example, in the below program, we have removed address operator '&' in assignment. We have also changed function call by removing *, the program still works.

```
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
   printf("Value of a is %d\n", a);
}

int main()
{
   void (*fun_ptr)(int) = fun;  // & removed
    fun_ptr(10);  // * removed
    return 0;
}
```

Output:

Value of a is 10

4) Like normal pointers, we can have an array of function pointers. Below example in point 5 shows syntax for array of pointers.

5) Function pointer can be used in place of switch case. For example, in below program, user is asked for a choice between 0 and 2 to do different tasks.

Example 7.8.2

```
#include <stdio.h>
void add(int a, int b)
{
   printf("Addition is %d\n", a+b);
}
void subtract(int a, int b)
{
   printf("Subtraction is %d\n", a-b);
}
void multiply(int a, int b)
{
   printf("Multiplication is %d\n", a*b);
}
```

```
int main()
{
    // fun_ptr_arr is an array of function pointers
    void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
    unsigned int ch, a = 15, b = 10;

    printf("Enter Choice: 0 for add, 1 for subtract and 2 "
            "for multiply\n");
    scanf("%d", &ch);

    if (ch > 2) return 0;

    (*fun_ptr_arr[ch])(a, b);

    return 0;
}
```

Output:
Enter Choice: 0 for add, 1 for subtract and 2 for multiply
2
Multiplication is 150

6) Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function. For example, consider the following C program where wrapper() receives a void fun() as parameter and calls the passed function.

Example 7.8.3

```
// A simple C program to show function pointers as parameter
#include <stdio.h>

// Two simple functions
void fun1() { printf("Fun1\n"); }
void fun2() { printf("Fun2\n"); }

// A function that receives a simple function
// as parameter and calls the function
void wrapper(void (*fun)())
{
    fun();
}

int main()
{
    wrapper(fun1);
    wrapper(fun2);
    return 0;
}
```

This point in particular is very useful in C. In C, we can use function pointers to avoid code redundancy. For example a simple qsort() function can be used to sort arrays in ascending order or descending or by any other order in case of array of structures. Not only this, with function pointers and void pointers, it is possible to use qsort for any data type.

```
// An example for qsort and comparator
#include <stdio.h>
#include <stdlib.h>

// A sample comparator function that is used
// for sorting an integer array in ascending order.
```

```
// To sort any array for any other data type and/or
// criteria, all we need to do is write more compare
// functions.  And we can use the same qsort()
int compare (const void * a, const void * b)
{
  return ( *(int*)a - *(int*)b );
}

int main ()
{
  int arr[] = {10, 5, 15, 12, 90, 80};
  int n = sizeof(arr)/sizeof(arr[0]), i;

  qsort (arr, n, sizeof(int), compare);

  for (i=0; i<n; i++)
     printf ("%d ", arr[i]);
  return 0;
}
```

Output:

5 10 12 15 80 90

7.9 DYNAMIC MEMORY ALLOCATION

The process of allocating memory during program execution is called dynamic memory allocation. C language offers 4 dynamic memory allocation functions. They are,

1. malloc()
2. calloc()
3. realloc()
4. free()

Function	Syntax
malloc ()	malloc (number *sizeof(int));
calloc ()	calloc (number, sizeof(int));
realloc ()	realloc (pointer_name, number * sizeof(int));
free ()	free (pointer_name);

7.9.1 MALLOC() FUNCTION

- malloc () function is used to allocate space in memory during the execution of the program.
- malloc () does not initialize the memory allocated during execution. It carries garbage value.
- malloc () function returns null pointer if it couldn't able to allocate requested amount of memory.

EXAMPLE 7.9.1 : PROGRAM FOR MALLOC() FUNCTION IN C

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

int main()
{
    char *mem_allocation;
    /* memory is allocated dynamically */
    mem_allocation = malloc( 20 * sizeof(char) );
    if( mem_allocation== NULL )
    {
      printf("Couldn't able to allocate requested memory\n");
    }
    else
    {
      strcpy( mem_allocation,"Problem Solving Using C");
    }
    printf("Dynamically allocated memory content : " \
          "%s\n", mem_allocation );
    free(mem_allocation);
}
```

OUTPUT:
Dynamically allocated memory content : Problem Solving Using C

7.9.2. CALLOC() FUNCTION IN C:

- calloc () function is also like malloc () function. But calloc () initializes the allocated memory to zero. But, malloc() doesn’t.

EXAMPLE 7.9.2: PROGRAM FOR CALLOC() FUNCTION IN C:

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

int main()
{
    char *mem_allocation;
    /* memory is allocated dynamically */
    mem_allocation = calloc( 20, sizeof(char) );
    if( mem_allocation== NULL )
    {
      printf("Couldn't able to allocate requested memory\n");
    }
    else
    {
       strcpy( mem_allocation," Problem Solving Using C ");
    }
       printf("Dynamically allocated memory content   : " \
             "%s\n", mem_allocation );
       free(mem_allocation);
}
```

OUTPUT:
Dynamically allocated memory content : Problem Solving Using C

7.9.3 REALLOC() FUNCTION IN C:

- realloc () function modifies the allocated memory size by malloc () and calloc () functions to new size.
- If enough space doesn't exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

7.9.4. FREE() FUNCTION IN C:

- free () function frees the allocated memory by malloc (), calloc (), realloc () functions and returns the memory to the system.

EXAMPLE 7.9.3: PROGRAM FOR REALLOC() AND FREE() FUNCTIONS IN C:

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

int main()
{
   char *mem_allocation;
   /* memory is allocated dynamically */
   mem_allocation = malloc( 20 * sizeof(char) );
   if( mem_allocation == NULL )
   {
      printf("Couldn't able to allocate requested memory\n");
   }
   else
   {
     strcpy( mem_allocation," Problem Solving Using C ");
   }
   printf("Dynamically allocated memory content  : " \
         "%s\n", mem_allocation );
   mem_allocation=realloc(mem_allocation,100*sizeof(char));
   if( mem_allocation == NULL )
   {
      printf("Couldn't able to allocate requested memory\n");
   }
   else
   {
      strcpy( mem_allocation,"space is extended upto " \
                        "100 characters");
   }
   printf("Resized memory : %s\n", mem_allocation );
   free(mem_allocation);
}
```

OUTPUT:

Dynamically allocated memory content : Problem Solving Using C
Resized memory : space is extended upto 100 characters

7.9.5 DIFFERENCE BETWEEN STATIC MEMORY ALLOCATION AND DYNAMIC MEMORY ALLOCATION IN C:

Static memory allocation	Dynamic memory allocation
In static memory allocation, memory is allocated while writing the C program. Actually, user requested memory will be allocated at compile time.	In dynamic memory allocation, memory is allocated while executing the program. That means at run time.
Memory size can't be modified while execution. Example: array	Memory size can be modified while execution. Example: Linked list

7.9.6 DIFFERENCE BETWEEN MALLOC() AND CALLOC() FUNCTIONS IN C:

malloc()	calloc()
It allocates only single block of requested memory	It allocates multiple blocks of requested memory
int *ptr;ptr = malloc(20 * sizeof(int));For the above, 20*4 bytes of memory only allocated in one block. Total = 80 bytes	int *ptr;Ptr = calloc(20, 20 * sizeof(int));For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory. Total = 1600 bytes
malloc () doesn't initializes the allocated memory. It contains garbage values	calloc () initializes the allocated memory to zero
type cast must be done since this function returns void pointer int *ptr;ptr = (int*)malloc(sizeof(int)*20);	Same as malloc () function int *ptr;ptr = (int*)calloc(20, 20 * sizeof(int));

7.10 Summary

- A **pointer** is a variable whose value is the address of another variable.
- '&' is called reference operator. It gives the address of a variable.
- There is another operator that gets you the value from the address, it is called a dereference operator (*).
- A pointer that is assigned NULL is called a **null** pointer.
- When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value.
- A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value.

- Pointer variables can be compared provided both variables point to objects of the same data type.
- When performing addition on a pointer, size of a pointer increments by no. of storage units and not by no. of bytes.
- Accessing array elements by pointers is **always** faster than accessing them by subscripts.
- The process of allocating memory during program execution is called dynamic memory allocation.
- malloc () function is used to allocate space in memory during the execution of the program.
- calloc () function is also like malloc () function, but calloc () initializes the allocated memory to zero.
- realloc () function modifies the allocated memory size by malloc () and calloc () functions to new size.
- free () function frees the allocated memory by malloc (), calloc (), realloc () functions and returns the memory to the system.

7.11 Questions for Exercises

1. if **int s[5]** is a one-dimensional array of integers, which of the following refers to the third element in the array?
   i. *(s + 2)
   ii. *(s + 3)
   iii. s + 3
   iv. s + 2
2. How is a pointer variable being declared? What is the purpose of data type included in the pointer declaration?
3. What is meant by array of pointers? What is pointer to pointer? Explain with examples.
4. How the indirection operator can be used to access a multidimensional array element?
5. Write short notes on array of pointers.
6. Write a program to copy the contents of one array into another in the reverse order using pointer.
7. Find the smallest number in an array using pointers.
8. Write a program to reverse a string using pointers.
9. What do you mean by dynamic memory allocation? explain all its functions.
10. What do you mean by pointer to function? Explain.

7.12 Suggested Readings

- *Programming in C*, Third Edition, Stephen G. Kochan
- *Let Us C*, Fifth Edition, Yashavant P. Kanetkar
- *The C programming language*, Brain W Kernighan, Dennis M Ritchie, PHI.
- *Programming with C*, Second Edition, Byron Gottfried, Tata McGraw Hill, 2003.
- *C. The Complete Reference*, Fourth Edition, Herbert Schildt, Tata McGraw Hill, 2002.
- *The C Primer*, Leslie Hancock, Morris Krieger, Mc Gravy Hill, 2013.