CHAPTER 6

DATA STRUCTURES

These refer to **groups of data elements that are organized in a single unit so that they can be used more efficiently** as compared to the simple data types such as integers and strings. An example of a data structure is the array. Ordinary variables store one value at a time while an array will store more than one value at a time in a single variable name.

Data structures are important for grouping sets of similar data together and passing them as one. For example, if you have a method that prints a set of data but you don't know when writing the procedure how large that set is going to be, you could use an array to pass the data to that method and loop through it.
Data structures can be **classified** using various criteria.

a) Linear

In linear data structures, values are arranged in linear fashion. A linear data structure traverses the data elements sequentially. The elements in the structure are adjacent to one another other and every element has exactly two neighbour elements to which it is connected. Arrays, linked lists, stacks and queues are examples of linear data structures.

b) Non-Linear

The data values in this structure are not arranged in order but every data item is attached to several other data items in a way that is specific for reflecting relationships. Tree, graph, table and sets are examples of non-linear data structures.

c) Homogenous

In this type of data structures, values of the same types of data are stored, as in an array.

d) Non-homogenous

In this type of data structures, data values of different types are grouped, as in structures and classes.

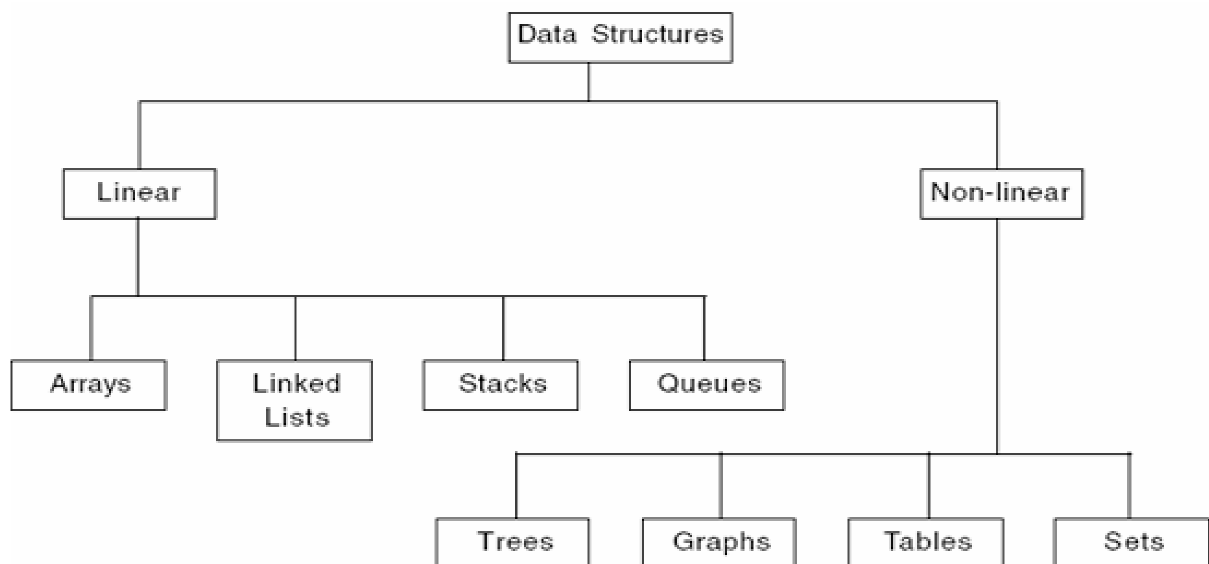
e) Dynamic

In dynamic data structures such as references and pointers, size and memory locations can be changed during program execution. These data structures can grow and shrink during execution.

f) Static

With a static data structure, the size of the structure is fixed. Static data structures such as arrays are very good for storing a well-defined number of data items.

www.knecnotes.co.ke



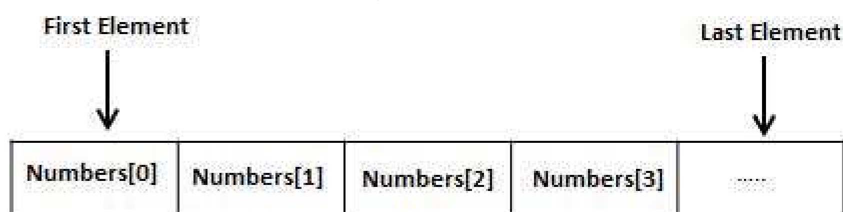
ARRAYS

An array is a named list of elements, all with the same data type. It is better defined as a consecutive group of memory locations all of which have the same name and the same data type. Arrays store a fixed-size sequential collection of elements of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as *numbers* and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

www.knecnotes.co.ke



DECLARING ARRAYS

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

type arrayName [arraySize];

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement:

double balance[10];

Now *balance* is a variable array which is sufficient to hold up to 10 double numbers.

INITIALIZING ARRAYS

You can initialize an array in C either one by one or using a single statement as follows:

double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array:

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

```
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

You will create exactly the same array as you did in the previous example.

```
balance[4] = 50.0;
```

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th ie. last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representation of the same array we discussed above:

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

www.kneunotes.co.ke

ACCESSING ARRAY ELEMENTS

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

double salary = balance[9];

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays:

```
#include <stdio.h>

int main ( )
{
    int n[ 10 ]; /* n is an array of 10 integers */
    int i,j;
```

```
    /* initialize elements of array n to 0 */
    for ( i = 0; i < 10; i++ )
    {
        n[ i ] = i + 100; /* set element at location i to i
+ 100 */
    }

    /* output each array element's value */
    for (j = 0; j < 10; j++ )
    {
        printf("Element[%d] = %d\n", j, n[j] );
    }

    return 0;
}
```

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

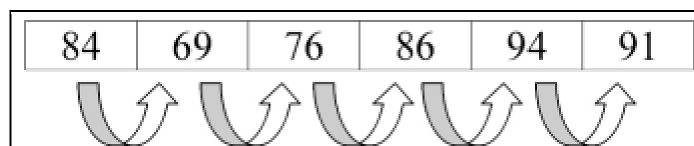
```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

www.knecnotes.co.ke

SORT TECHNIQUES

Bubble Sort

In the **bubble sort,** as elements are sorted they gradually "bubble" (or rise) to their proper location in the array, like bubbles rising in a glass of soda. The bubble sort repeatedly compares **adjacent elements** of an array. The first and second elements are compared and swapped if out of order. Then the second and third elements are compared and swapped if out of order. This sorting process continues until the last two elements of the array are compared and swapped if out of order.



When this first pass through the array is complete, the bubble sort returns to elements one and two and starts the process all over again.

The table below follows an array of numbers before, during, and after a bubble sort

fordescending order. A "pass" is defined as one full trip through the array comparing and if necessary, swapping, **adjacent** elements. Several passes have to be made through the array before it is finally sorted

Array at beginning:	84	69	76	86	94	91
After Pass #1:	84	76	86	94	91	69
After Pass #2:	84	86	94	91	76	69
After Pass #3:	86	94	91	84	76	69
After Pass #4:	94	91	86	84	76	69
After Pass #5 (done):	94	91	86	84	76	69

The bubble sort is an easy algorithm to program, but it is slower than many other sorts. With a bubble sort, it is always necessary to make one final "pass" through the array to check to see that no swaps are made to ensure that the process is finished. In actuality, the process is finished before this last pass is made.

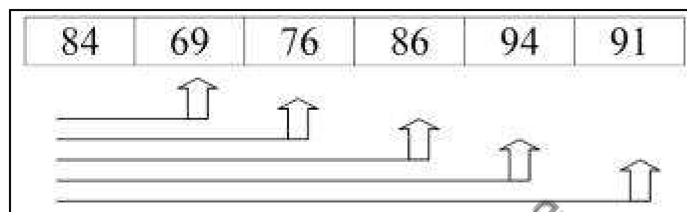
```
// Bubble Sort Function for Descending Order
#include<stdio.h>
main()
{
int control , control2, marks, total=0, temp;float meanmark;
int allmarks[5];

for (control = 0; control <= 4; control++)
{
        printf("Please enter student's marks:");
        scanf("%d", & marks);
        allmarks[control]=marks;
        total = total + marks;
}
meanmark = (float) total/control;

for (control = 0; control < 4; control++) {
        for (control2 = 0; control2 < 4; control2++) {
                if (allmarks[control2] > allmarks[control2+1])
                {
                        temp = allmarks[control2];
                        allmarks[control2]= allmarks[control2+1];
                        allmarks[control2+1] = temp;
                }
        }
}

 printf("\nThe sorted list of marks is:\n");

for (control=0; control<=4;control++)
{
printf("%d\n", allmarks[control]);
}
```

www.knecnotes.co.ke

```
printf("\nThe total marks is %d\n", total);
printf("Mean marks is %f\n", meanmark);
}
```

Exchange Sort

The **exchange sort** is similar to its cousin, the bubble sort, in that it compares elements of the array and swaps those that are not in their proper positions. (Some people refer to the "exchange sort" as a "bubble sort".) The difference between these two sorts is the manner in which they compare the elements. **The exchange sort compares the first element with each following element of the array, making any necessary swaps.**



When the first pass through the array is complete, the exchange sort then takes the second element and compares it with each following element of the array swapping elements that are out of order. This sorting process continues until the entire array is ordered.

www.knecnotes.co.ke

Let's examine our same table of elements again using an exchange sort for descending order. Remember, a "pass" is defined as one full trip through the array comparing and if necessary, swapping elements.

Array at beginning:	84	69	76	86	94	91
After Pass #1:	94	69	76	84	86	91
After Pass #2:	94	91	69	76	84	86
After Pass #3:	94	91	86	69	76	84
After Pass #4:	94	91	86	84	69	76
After Pass #5 (done):	94	91	86	84	76	69

The exchange sort, in some situations, is slightly more efficient than the bubble sort. It is not necessary for the exchange sort to make that final complete pass needed by the bubble sort to determine that it is finished.

//Exchange Sort Function for Descending Order

```
#include <stdio.h>
void main()
```

```
{
    int i, j;
    int temp;   // holding variable
    int num[5];

            //intialize array
            for(i =0; i<=4; i++){
            printf("Enter a number:");
            scanf("%d",&num[i]);
            }
           //sort array
    for (i=0; i< (4); i++)    // element to be compared
    {
        for(j = (i+1); j < 5; j++)   // rest of the elements
        {
              if (num[i] < num[j])          // descending order
             {
                     temp= num[i];          // swap
                     num[i] = num[j];
                     num[j] = temp;
             }
        }
    }
//print sorted array
            printf("\nSorted array:\n");

            for(i =0; i<=4; i++){
            printf("\t%d\n", num[i]);
                      }
    return;
}
```

www.knecnotes.co.ke

Selection Sort

The **selection sort** is a combination of searching and sorting.

During each pass, the unsorted element with the smallest (or largest) value is moved to its proper position in the array.
The number of times the sort passes through the array is one less than the number of items in the array. In the selection sort, the inner loop finds the next smallest (or largest) value and the outer loop places that value into its proper location.

Let's look at our same table of elements using a selection sort for descending order. Remember, a "pass" is defined as one full trip through the array comparing and if necessary, swapping elements.

Array at beginning:	84	69	76	86	94	91
After Pass #1:	84	91	76	86	94	69

After Pass #2:	84	91	94	86	76	69
After Pass #3:	86	91	94	84	76	69
After Pass #4:	94	91	86	84	76	69
After Pass #5 (done):	94	91	86	84	76	69

While being an easy sort to program, the selection sort is one of the least efficient. The algorithm offers no way to end the sort early, even if it begins with an already sorted list.

// Selection Sort Function for Descending Order

```
void main()
{
    int i, j, first, temp;
    int num[5]
    for (i= 4; i > 0; i--)
    {
        first = 0;              // initialize to subscript of first element
        for (j=1; j<=i; j++)    // locate smallest between positions 1 and i.
        {
            if (num[j] < num[first])
            first = j;
        }
        temp = num[first];   // Swap smallest found with element in position i.
        num[first] = num[i];
        num[i] = temp;
    }
    return;
}
```

www.kneconotes.co.ke

Shell Sort

The **shell sort** is named after its inventor D. L. Shell. Instead of comparing adjacent elements, like the bubble sort, the shell sort repeatedly compares elements that are a certain distance away from each other (*d* represents this distance). The value of *d* starts out as half the input size and is halved after each pass through the array. The elements are compared and swapped when needed. The equation $d= (N + 1) / 2$ is used. Notice that only integer values are used for *d* since integer division is occurring.

Let's look at our same list of values for descending order with the shell sort. Remember, a "pass" is defined as one full trip through the array comparing and if necessary, swapping elements.

Array at beginning:	84	69	76	86	94	91	*d*
After Pass #1:	86	94	91	84	69	76	3
After Pass #2:	91	94	86	84	69	76	2

After Pass #3:	94	91	86	84	76	69	1
After Pass #4 (done):	94	91	86	84	76	69	1

First Pass: $d = (6 + 1) / 2 = 3$. Compare 1st and 4th , 2nd and 5th, and 3rd and 6th items since they are 3 positions away from each other))
Second Pass: value for d is halved $d = (3 + 1) / 2 = 2$. Compare items two places away such as 1st and 3rd ……
Third Pass: value for d is halved $d = (2 + 1) / 2 = 1$. Compare items one place away such as 1st and 2nd …..
Last Pass: sort continues until $d = 1$ **and** the pass occurs without any swaps.

This sorting process, with its comparison model, is an efficient sorting algorithm.

//Shell Sort Function for Descending Order

```
void main()
{
    int I,d , temp,  length[5];
     while( (d > 1))      // boolean flag (true when not equal to 0)
    {
        d = (d+1) / 2;
        for (i = 0; i < (5 - d); i++)
      {
            if (num[i + d] > num[i])
           {
                temp = num[i + d];      // swap positions i+d and i
                num[i + d] = num[i];
                num[i] = temp;
                flag = 1;               // tells swap has occurred
           }
        }
    }
    return;
}
```

www.kneonotes.co.ke

Quick Sort

The **quicksort** is considered to be very efficient, with its "divide and conquer" algorithm. This sort starts by dividing the original array into two sections (partitions) based upon the value of the first element in the array. Since our example sorts into descending order, the first section will contain all the elements greater than the first element. The second section will contain elements less than (or equal to) the first element. It is possible for the first element to end up in either section.

Let's examine our same example

Array at beginning:	84	69	76	86	94	91
= 1st partition	86	94	91	84	69	76
= 2nd partition	94	91	86	84	69	76
	94	91	86	84	69	76

	94	91	86	84	69	76
Done:	94	91	86	84	76	69

This sort uses recursion - the process of "calling itself". Recursion will be studied at a later date.

//Quick Sort Functions for Descending Order
// (2 Functions)

```
void main()
{
    // top = subscript of beginning of array
    // bottom = subscript of end of array

   int middle;
    if (top < bottom)
   {
        middle = partition(num, top, bottom);
        quicksort(num, top, middle);   // sort first section
        quicksort(num, middle+1, bottom);    // sort second section
    }
    return;
}
```

www.knecnotes.co.ke

//Function to determine the partitions

// partitions the array and returns the middle subscript

```
int main()
{
    int x = array[top];
    int i = top - 1;
    int j = bottom + 1;
    int temp;
    do
    {
        do
        {
              j - -;
        }while (x >array[j]);

        do
       {
              i++;
        } while (x <array[i]);

        if (i < j)
       {
              temp = array[i];
              array[i] = array[j];
```

```
            array[j] = temp;
        }
    }while (i < j);
    return j;          // returns middle subscript
}
```

Merge Sort

The **merge sort** combines two **sorted** arrays into one larger sorted array.
As the diagram below shows, Array A and Array B merge to form Array C.

Arrays to be merged **MUST be SORTED FIRST!!**

Be sure to declare Array C in main( ) and establish its size.

Example: Ascending Order
Array A: {7. 12}
Array B: {5, 7, 8}
Array C: {5, 7, 7, 8, 12} after merge

Here is how it works: The first element of array A is compared with the first element of array B. If the first element of array A is smaller than the first element of array B, the element from array A is moved to the new array C. The subscript of array A is now increased since the first element is now set and we move on.

If the element from array B should be smaller, it is moved to the new array C. The subscript of array B is increased. This process of comparing the elements in the two arrays continues until either array A or array B is empty. When one array is empty, any elements remaining in the other (non-empty) array are "pushed" into the end of array C and the merge is complete.

//Function to merge two pre-sorted arrays

```
void main()
{
    int indexA = 0;     // initialize variables for the subscripts
    int indexB = 0;
    int indexC = 0;
    Int arrayC[5];

    while((indexA < 5) && (indexB < 5)
    {

        if (arrayA[indexA] < arrayB[indexB])
        {
            arrayC[indexC] = arrayA[indexA];
            indexA++;    //increase the subscript
        }
        else
        {
            arrayC[indexC] = arrayB[indexB];
            indexB++;    //increase the subscript
```

www.kenyanotes.co.ke

```
        }
        indexC++;       //move to the next position in the new array
    }
    // Move remaining elements to end of new array when one merging array is empty
    while (indexA < 5)
    {
        arrayC[indexC] = arrayA[indexA];
        indexA++;
        indexC++;
    }
    while (indexB < 5)
    {
        arrayC[indexC] = arrayB[indexB];
        indexB++;
        indexC++;
    }
    return;
}
```

SEARCHING ARRAYS

When working with arrays, it is often necessary to perform a search or "lookup" to determine whether an array contains a value that matches a certain key value The process of locating a particular element value in an array is called searching. There are two types of search mechanisms: **serial/linear search** and **binary search**

a) Serial Search

The technique used here is called a **serial search**, because the integer elements of the array are compared one by one to the user input being looked for (userValue) until either a match is found or all elements of the array are examined without finding a match.

In the code below, if a match is found, the text "There is a match" is printed on the form and the execution of the procedure is terminated (Exit Sub). If no match is found, the program exits the loop and prints the text "No match found".

```
#include <stdio.h>


int main()

{

    int array[5]={10,7,8,2,5}, searchvalue, c;


    printf("\tEnter the number to search: ");

    scanf("%d", &searchvalue);


    for (c = 0; c < 5; c++)

    {

        if (array[c] == searchvalue)      // if required element
found

        {
```

www.knecnotes.co.ke

```
            printf("\n\t%d is present at location %d.\n",
searchvalue, c+1);

            break;

        }

       }

    if (c == 5) // if looped more than 5 times ie 6 times

        printf("\n\t%d is not present in the array.\n",
searchvalue);


    return 0;

}
```

Binary Search

Binary search uses the concept of splitting your searchable array in two, discarding the half that does not have the element for which you are looking.

You place your items in an array and sort them. Then you simply get the middle element and test if it is <, >, or = to the element for which you are searching. If it is less than, you discard the greater half, get the middle index of the remaining elements and do it again. Binary search divides your problem in half every time you execute your loop.

www.kenyanotes.co.ke

```
#include <stdio.h>

int main()
{
   int c, first, last, middle, n, search, array[100];

   printf("Enter number of elements\n");
   scanf("%d",&n);

   printf("Enter %d integers\n", n);

   for ( c = 0 ; c < n ; c++ )
      scanf("%d",&array[c]);

   printf("Enter value to find\n");
   scanf("%d",&search);

   first = 0;
   last = n - 1;
   middle = (first+last)/2;

   while( first <= last )
   {
      if ( array[middle] < search )
         first = middle + 1;
      else if ( array[middle] == search )
      {
         printf("%d found at location %d.\n", search, middle+1);
         break;
      }
      else
         last = middle - 1;

      middle = (first + last)/2;
```

```
    }
    if ( first > last )
       printf("Not found! %d is not present in the list.\n", search);

    return 0;
}
```

LINKED LISTS

A linked list is a dynamic data structure whose length can be increased or decreased at run time.

How Linked lists are different from arrays? Consider the following points :

- An array is a static data structure. This means the length of array cannot be altered at run time. While, a linked list is a dynamic data structure.
- In an array, all the elements are kept at consecutive memory locations while in a linked list the elements (or nodes) may be kept at any location but still connected to each other.

When to prefer linked lists over arrays? Linked lists are preferred mostly when you don't know the volume of data to be stored. For example, In an employee management system, one cannot use arrays as they are of fixed length while any number of new employees can join. In scenarios like these, linked lists (or other dynamic data structures) are used as their capacity can be increased (or decreased) at run time (as an when required).

www.knecnotes.co.ke

How linked lists are arranged in memory?

Linked list basically consists of memory blocks that are located at random memory locations. Linked lists are connected through pointers.

POINTERS

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 you can use 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 * you used to declare a pointer is the

same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
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.

HOW TO USE POINTERS?

There are 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. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

```
#include <stdio.h>

int main ()
{
   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */

   ip = &var;  /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

NULL Pointers in C

www.knecnotes.co.ke

It is always a good practice to assign a NULL value to a pointer variable in case you do not have 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;
}
```

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

```
The value of ptr is 0
```

On 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.

www.knecnotes.co.ke

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 */
```

C STRINGS

In C, one or more characters enclosed between double quotes is called a string. C does not have built-in string data type. Instead, C supports strings using one-dimensional arrays. A string is defined as a *null terminated* array i.e. \0. This means that you must define the array that is going to hold a string to be one byte larger than the largest string it is going to hold, in order to make room for the null.

To read a string from the keyboard, you must use another of C's standard library functions, **gets( )** , which requires the **stdio.h** header file. The gets ( ) function reads characters until you presss **<ENTER>.** The carriage return is not stored, but it is replaced by a null, wich terminates the string. E.g.

```
#include<stdio.h>
Main ( )
Char str [80];
Int I;
Printf (ËNter a string: \n");
```

```
gets(str);

for (I = 0; str[i]; i++)
printf("%c", str[i]);
}
```

The following declaration and initialization create a string consisting of the word **"Hello"**. To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word **"Hello"**.

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Initialization of strings

In C, string can be initialized in a different number of ways.

```
char c[]="abcd";
     OR,
char c[5]="abcd";
     OR,
char c[]={'a','b','c','d','\0'};
     OR;
char c[5]={'a','b','c','d','\0'};
```

c[0]	c[1]	c[2]	c[3]	c[4]
a	b	c	d	\0

www.knecnotes.co.ke

String can also be initialized using pointers

```
char *c="abcd";
```

Reading Strings from user.

Reading words from user.

```
char c[20];
scanf("%s",c);
```

String variable *c* can only take a word. It is beacause when white space is encountered, the `scanf()` function terminates.

Write a C program to illustrate how to read string from terminal.

```
#include <stdio.h>
int main(){
    char name[20];
    printf("Enter name: ");
    scanf("%s",name);
    printf("Your name is %s.",name);
    return 0;
}
```

Output

```
Enter name: Dennis Ritchie
Your name is Dennis.
```

Here, program will ignore Ritchie because, `scanf()` function takes only string before the white space.

C supports a wide range of functions that manipulate null-terminated strings:

S.N.	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

www.knecnotes.co.ke

The C library function **int strcmp(const char *str1, const char *str2)** compares the string pointed to by **str1** to the string pointed to by **str2**.

Following is the declaration for strcmp() function.

```
int strcmp(str1, str2)
```

PARAMETERS

- **str1** -- This is the first string to be compared.
- **str2** -- This is the second string to be compared.

RETURN VALUE

This function returned values are as follows:

- if Return value if < 0 then it indicates str1 is less than str2
- if Return value if > 0 then it indicates str2 is less than str1
- if Return value if = 0 then it indicates str1 is equal to str2

Example

The following example shows the usage of strncmp() function.

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

int main ()
{
   char str1[15];
   char str2[15];
   int ret;


   strcpy(str1, "abcdef");
   strcpy(str2, "ABCDEF");

   ret = strcmp(str1, str2);

   if(ret > 0)
   {
      printf("str1 is less than str2");
   }
   else if(ret < 0)
   {
      printf("str2 is less than str1");
   }
   else
   {
      printf("str1 is equal to str2");
   }

   return(0);
}
```

www.knecnotes.co.ke

More Examples

1) C Program to Find the Length of a String

```
#include <stdio.h>
int main()
{
    char s[1000],i;
    printf("Enter a string: ");
    scanf("%s",s);
    for(i=0; s[i]!='\0'; ++i);
    printf("Length of string: %d",i);
    return 0;
}
```

Output

```
Enter a string: Programiz
Length of string: 9
```

2) Code to Concatenate Two Strings Manually

```
#include <stdio.h>
int main()
{
    char s1[100], s2[100], i, j;
```

```
    printf("Enter first string: ");
    scanf("%s",s1);
    printf("Enter second string: ");
    scanf("%s",s2);
    for(i=0; s1[i]!='\0'; ++i);  /* i contains length of string
s1. */
    for(j=0; s2[j]!='\0'; ++j, ++i)
    {
        s1[i]=s2[j];
    }
    s1[i]='\0';
    printf("After concatenation: %s",s1);
    return 0;
}
```

Output

```
Enter first string: lol
Enter second string: :)
After concatenation: lol:)
```

QUEUES

Queue is a specialized data storage structure (Abstract data type). Unlike arrays, access of elements in a Queue is restricted. It has two main operations enqueue and dequeue. Insertion in a queue is done using enqueue function and removal from a queue is done using dequeue function. An item can be inserted at the end ('rear') of the queue and removed from the front ('front') of the queue. It is therefore, also called First-In-First-Out (FIFO) list. Queue has five properties - capacity stands for the maximum number of elements Queue can hold, size stands for the current size of the Queue, elements is the array of elements, front is the index of first element (the index at which we remove the element) and rear is the index of last element (the index at which we insert the element).

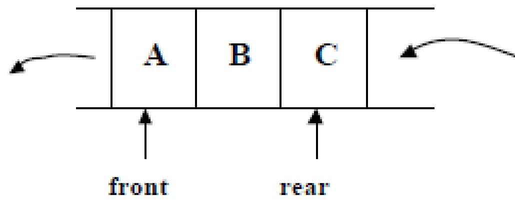
www.kenyanotes.co.ke

Primitive operations

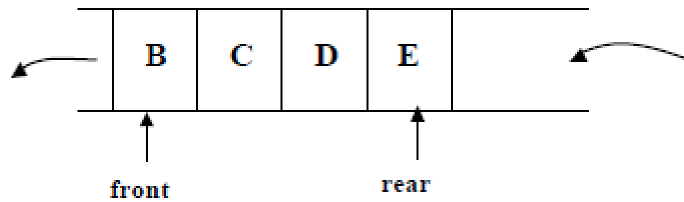
a) enqueue (q, x): inserts item **x** at the rear of the queue **q**
b) x = dequeue (q): removes the front element from **q** and returns its value.
c) isEmpty(q) : true if the queue is empty, otherwise false.

Example

enqueue(q, 'A');
enqueue(q, 'B');
enqueue(q, 'C');
x = dequeue(q);
enqueue(q, 'D');
enqueue(q, 'E');



x= dequeue (q) -> x= 'A'



STACKS

A stack is a data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack; the only element that can be removed is the element that was at the top of the stack. Consequently, a stack is said to have "first in last out" behavior (or "last in, first out"). The first item added to a stack will be the last item removed from a stack.

www.knecnotes.co.ke