1. ARRAYS, STRINGS AND POINTERS

1.1 Introduction to Strings

A string is a sequence of characters (array), with the last character being the NUL character (`\0`).

> For example, the array of 6 characters: `I n d i a \0` is a string of length 5 (the NUL character is not counted as part of the length). Had the same array been without the terminating \0, it would not be called a string, but simply a character array.

Because of this extra \0 character (whose ASCII value is 0) that signals the end of strings, they can be treated in special ways that normal arrays cannot be treated.

1.2 String variables, string constants and declarations

`char word[40];` declares an array of forty characters and can be treated as a string if a \0 is one of the elements.

Consider these three array declarations:

```
char arr[] = "Hello";
char brr[7] = {'H', 'e', 'l', 'l', 'o'};
char crr[] = {'H', 'e', 'l', 'l', 'o'};
```

The entity `arr` is a string, because the variable `arr` is being initialized with the string constant "`Hello`" at the time of declaration. (This is similar to saying `int i = 4;` insofar initializing an integer variable with an integer constant.)

Similarly, `brr` is also a string, for if even one element of an array is initialized (at the time of declaring the array), the remainder of the uninitialized elements are automatically set to 0.

The third array, `crr` is not a string, because there is no \0 assignment in the declaration, and only 5 bytes are set aside in memory. Hence, crr is just a character array.

The following three entities have different meanings: `4`, `'4'`, `"4"`
4 is an integer constant, '4' is a character constant (with an ASCII value of 52), while "4" is a string constant.

1.3 Getting input into string variables and to printing out strings

The conversion specifier "%s" is used by both `scanf()` and `printf()` functions to get string input, and give string output, respectively. Consider this piece of code:

```
char word1[100];
printf("Enter the word: ");
scanf("%s", word1);
printf("The word you entered is: %s\n", word1);
```

Note that `scanf()` expects an argument that is an address (pointer). Since the name of the array is its base address, you don't need to have its name preceded with the address of operator (&).

The %s specifier in the `scanf()` function will take input until the first whitespace character, [i.e., a space, \n (newline) or \t (tab) character] is encountered. So, how to take a line of input?

```
scanf("%[^\n]", word1);
```

By specifying the character class as the format specifier, where you state that all characters till \n (the caret symbol denotes negation of the character class) should be accepted, you can now take a line of input.

The character class can be used in `scanf()` in other usual ways such as `%[A-Za-z0-9]` to accept a string of only alphanumeric characters, `%[aeiouAEIOU,]` to accept a string of vowels and a comma, and so on. In such cases, the `scanf()` stops accepting input at the first instance of the condition being violated. While no further input is taken into the string, the remaining input continues to stay in the input buffer! This may cause problems with subsequent calls to an input function like `scanf()` and `getchar()`. It is important to keep this in mind when writing code.

For instance, when receiving input for a line of text using the statement `scanf("%[^\n]", word1);` or even when using `scanf()` to get a numeric input, e.g., `scanf("%d", &N)`, the \n character entered in the keyboard continues to stay in the input buffer, available for the subsequent call to `scanf()` potentially causing problems. For example, an empty string gets stored in the array `word1` when a subsequent `scanf("%[^\n]", word1);` statement in encountered. One way to remedy this issue is by including an input statement to eat away the residual \n still present in the input buffer stream:

```
scanf("%[^\n]", word1);
getchar();   /* an empty getchar() to dissipate the \n */
```

LABORATORY SESSION #8

Exercise 8-1: Write a program that accepts a line of text as a string from the user and prints out only the non-whitespace characters of it. You may assume a line contains at most 99 characters.

Next, modify the program so it works for N lines of text, where N is a number between 1 and 10 input by the user. You can take one line, process it and print the output before accepting the next line of input into the same array. **Do not** use a 2D array, but just a nested loop.

Omit the empty `getchar()` statement after the `scanf()` and observe what happens. You should understand how to handle the input buffer stream properly, otherwise your program is likely to show unexpected behaviour.

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

printf("How many lines of text? ");
do { scanf("%d", &N); } while (N < 1 && N > 10);
getchar();   /* to dissipate the \n character after the number entered */

for (i = 1; i <= N; ++i)
{
  scanf("%[^\n]", line);
  getchar();
  for (j = 0; line[j]; ++j)
  {
       if (line[j] == ' ' || line[j] == '\t')
                    continue;    /* if whitespace, don't print it */
       putchar(line[j]);
  }
  putchar('\n');
}
return 0;
}
```

1.4 String Functions

The essence of treating a character array as a string is best put to use by string functions (available in the standard C library). Use of these functions requires the programmer to include the `<string.h>` header file in the program. The prototypes and use of some of the commonly used string library functions are given below:

```
size_t strlen(const char *s);    // size_t is long unsigned int on gcc
```

Returns the length (number of bytes, excluding \0) of the string **s**.

```
char *strcpy(char *dest, const char *src);
```

Copies the string **src** to **dest**, returning a pointer to the start of **dest**.

```
int strcmp(const char *s1, const char *s2);
```

Compares the strings **s1** with **s2**. Returns 0 if **s1** is identical to **s2**, a negative value if **s1** < **s2**, and a positive value if **s1** > **s2**.

```
char *strcat(char *dest, const char *src);
```

Appends the string **src** to the string **dest**, returning a pointer to **dest**.

```
char *strchr(const char *s, int c);
```

Returns a pointer to the first occurrence of the character **c** in the string **s**.

```
char *strstr(const char *haystack, const char *needle);
```

Finds the first occurrence of the substring **needle** in the string **haystack**, and returns a pointer to the found substring, or NULL if the substring is not found.

Exercise 8-2: Type `man string` on the shell prompt to see the manual page entry of the collection of string functions available. Look up the individual manual page entries of these four functions specifically: `strlen()`, `strcmp()`, `strcpy()`, `strcat()`. Write few lines of code to understand how these functions work. For instance, you may try these as part of your program:

```
char s1[50] = "BITS", s2[] = "Pilani", s3[50];
printf("%lu %lu\n", strlen(s1), sizeof(s1));
    /* It is easy to get confused between strlen() and sizeof() */
strcat(s1," ");
strcat(s1, s2);
strcat(s1, ", ");
```

```
strcpy(s3,"Pilani Campus");
strcat(s1, s3);
puts(s1);       /* Look up the man page of puts() to learn more */
/* comparing two strings… */
if (!strcmp(s2,s3)) /* strcmp() has returned 0 */
    puts("Both strings are the same!\n");
else
if (strcmp(s2,s3) > 1)
    printf("%s is greater than %s\n", s2, s3);
else
    printf("%s is greater than %s\n", s3, s2);
```

Exercise 8-3: In the mid-semester test you took last week, a pointer-based implementation of `strlen()` function was provided:

```
int my_str(char p[])
{
    char *q = p;
    while (*q)
      q++;
    return q-p;
}
```

Now, you implement the same function without using pointers. You may name this function as **`my_strlen1()`**.

Exercise 8-4: Implement a function **`convertLower()`** (whose prototype is given below) to convert a given string into one that contains entirely lower case letters.

```
void convertLower(char *);
```

The function performs the conversion on the string received as its argument. First implement your function using ASCII codes only. Next, use the library function `tolower()` to do the job. Remember to include the header `<ctype.h>`. You can learn more about this function by using the online manual page.

[*Note:* There are a suite of useful library functions like `tolower()` and `toupper()` available in the ctype library. Get to know them by typing man isalpha at the shell prompt. You should be familiar with the first 13 functions – starting from `isalnum()` and ending with `isblank()`, whose descriptions are also provided in the manual page, so you can use them in your programs when needed.]

Exercise 8-5: Write a function that checks to see if the string it takes as an argument is a palindrome or not.

(a) Write the first version of the function **`isPalind_1()`** using array notation, and the second version **`isPalind_2()`** using pointers.

(b) Now, modify one of the above two functions so that **`isPalind_3()`** that you write now uses a function **`strrev()`** whose definition is given below:

```
void strrev(char *a) /* function to reverse the given string */
{
  char *b, temp;
  b = a + strlen(a) - 1;  /* b is made to the last character preceding
                             the terminating \0 */
  for (  ; a < b ; a++, b--)
       {           /* swapping the contents pointed to by a and b */
         temp = *a;
         *a = *b;
         *b = temp;
       }
  return;
}
```

1.5 Arrays and Pointer Arithmetic

Arrays and pointers have a strong connection. The name of an array represents the base address of the array, and can be assigned to a pointer. Consider the following code segment:

```
char arr[13];     char *parr;       parr = arr;
```

Here, `arr` represents an array of 13 integer elements, and `arr` is the base address of the array, i.e., address of the 0th element. In other words, `arr` is equivalent to `&arr[0]`.

Delving on this further,

> **`arr[i]`** is the same as writing **`*( arr + i)`**. Both refer to the value stored at the address given by the expression arr + i * sizeof (char).
> **`&arr[i]`** is the same as writing **`arr + i`**. Both refer to the address of the i^{th} element of the array.

Because the name of an array is its base address, when an array is passed to a function, what is being passed is only the array's base address, and not the entire array.

```
void foo(char []);
int main()
{ char arr[100];
strcpy(arr,"BITS Pilani");
printf("%s\n", arr);
foo(arr);
```

```
}                                          printf("%p\n", ptr);
void foo(char ptr[])                       printf("%s\n", ptr);
{                                          return;
printf("Inside foo...");                   }
```

In the above example, the array `arr` (storing the string) is allocated 100 bytes of memory only in the `main()` function, and not in `foo()`. What is being passed is the base address of `arr` (i.e., the starting location of the array), and ***the value of this address*** is received by `foo()` as an argument and is allotted storage under the variable name `ptr`. Since what is being received is an address, `ptr` is treated as a pointer, and so 8 bytes are allotted for it as a local variable in `foo()`. Now, when you print the value of it in the function, the base address gets printed. And using this address, you can also retrieve the contents of the array, and hence print the string "BITS Pilani" from the function as well (just as you are able to do the same within the `main()` function).

Because what is being received is an address, the formal argument can also be received a pointer. So, `void foo(char ptr[])` could have also been written as `void foo(char *ptr)`. In fact, the compiler treats the array notation of a formal argument to a function only as a pointer. How do we know this is true?
`sizeof(ptr)` inside `foo()` will yield 8, which is the number of bytes allocated to a pointer, whereas `sizeof(arr)` inside `main()` will yield 100, which is the size of the array allocated.

However, a pointer is pointer, and an array is an array!
Consider the following declaration and initialization:

```
char arr[108];    char *p = arr;
```

While `p` can be used synonymously with `arr` e.g., `p[i]`, `strlen(p)`, `strcmp(p,name)` and the likes, `arr` cannot be used as a replacement for `p`. The best illustration is you can legally do `p++` to increment the address by one byte (since `p` is declared as a `char *`), but it is illegal (a syntax error) to increment the base address by saying `arr++`.

To understand pointers and strings better, the following implementations of the library function `strcpy()` may be useful:

```
char * my_strcpy1(char *dest, char *src) /* copies src into destination array */
{
```

```
	int i, len;
	len = strlen (src);
	for ( i = 0; i <= len; ++i)
		dest[i] = src[i];  /* copies everything till '\0', including it */
	return dest;
}

char * my_strcpy2(char *dest, char *src)
{
	int i = 0;
	while ((dest[i] = src[i]) != '\0')  i++;
			/* first copy to dest, then check for NUL character */
	return dest;
}

char my_strcpy2a(char *dest, char *src)
{
	int i = 0;
	while (dest[i] = src[i])  i++;  /* using ASCII value of '\0' is zero */
	return dest;
}

char my_strcpy3(char *dest, char *src) /* pointer version */
{
	char *old_dest = dest;  /* making a copy of dest */
	while (*dest = *src)
		{	dest++;	src++;}
	return old_dest;	/* return the base address of dest array */
}
char my_strcpy3a(char *dest, char *src) /* a compact, pointer version */
{
	char *old_dest = dest;  /* making a copy of dest */
	while (*dest++ = *src++) /* dereference, use the dereferenced values and
then increment pointers */
			;		/* null statement */
	return old_dest;
}
```

2. TWO-DIMENSIONAL ARRAYS AND ARRAYS OF POINTERS

2.1 Introduction

Since you are all familiar with matrices in mathematics, using 2D arrays will be very intuitive. *A 2D array is just an array of elements, each element being a one-dimensional array.*

A 1D array of 10 elements, each element of which is an integer is declared as:

```
int arr[10];
```

A 2D array of 10 elements, each element being an array of 4 integers is declared as:

```
int brr[10][4];
```

Memory allocated for `brr` is 10 * (4 * `sizeof` (`int`)) bytes. On the GCC you use in this course, since the size of an integer is 4 bytes, `brr` will be allocated 160 bytes of memory.

Though `brr` can now be treated as 10 x 4 matrix (10 rows and 4 columns), memory allocation is done linearly, as done for any other array. So, `brr[0]` would be first (16 bytes), followed by `brr[1]` and so on till `brr[9]`.

2.2 Getting data into a 2D array

The first way to get data is by initializing the elements when declaring the array, as shown here:

```
int mat1[2][4] = {{1,11,-2,6}, {2,4,-10,16}};
```

The above statement declares a 2D array of dimension 2x4, and makes the following initializations:

`mat1[0][0]` = 1	`mat1[1][0]` = 2
`mat1[0][1]` = 11	`mat1[1][1]` = 4
`mat1[0][2]` = -2	`mat1[1][2]` = -10
`mat1[0][3]` = 6	`mat1[1][3]` = 16

The following declaration initializes some elements (row 1 entirely, first two elements of row 2, first element of row 3) while the remaining are set to zero:

```
int mat2[4][4] = {{1,11,-2,6}, {-2,4}, {-5}};
```

The next declaration does not mention the number of rows, and lets the compiler calculate it automatically as 3, based on the initialization done:

```
int mat3[][4] = {{1,11,-2,6}, {-2,4}, {-5}};
```

The second way by which input can be obtained into a 2D array is by using `scanf()`. That is being done here by a user-defined function `matrixScan()` which takes the matrix as it argument:

```
void matrixScan(int s[ROWS][COLS])
          /* ROWS and COLS are symbolic constants #defined by user */
{ int i,j;
  for (i = 0; i < ROWS; i++)
    for (j = 0; j < COLS; j++)
          scanf("%d",&s[i][j]);
}
```

And here is a function that prints out the values of a 2D array in a matrix notation:

```
void matrixPrint(int s[][COLS])
{ int i,j;
  for (i = 0; i < ROWS; i++)
  {
    for (j = 0; j < COLS; j++)
          printf("%7d ",s[i][j]);
    putchar('\n');   /* to move to the next line after printing a row */
  }
}
```

Notice how the first dimension is not mentioned when the 2D array is being received as an argument to the function, because it is optional. The second dimension, however, is required to be mentioned (else, it is a compile-time error).

The function **`matrixAdd()`** adds matrices **`a`** and **`b`**, and stores the result in **`c`**:

```
void matrixAdd(int a[][COLS], int b[][COLS], int c[][COLS])
{
int i, j;
for (i = 0; i < ROWS; i++)
  for (j = 0; j < COLS; j++)
          c[i][j] = a[i][j] + b[i][j];
return;
}
```

We can also declare 2D arrays of characters like this:

```
char objects[4][3] = {{'m', 'a', 't'}, {'b', 'a', 't'}, {'n', 'e', 't'},
                      {'r', 'o', 'd'}, {'p', 'i', 'n'}};
```

Or like this:

```
char objects[][3] = {{'m', 'a', 't'}, {'b', 'a', 't'}, {'n', 'e', 't'},
                     {'r', 'o', 'd'}, {'p', 'i', 'n'}};
```

Not only is this method of declaring arrays cumbersome (writing each character), but is also of limited utility. Rather, 2D arrays can be used to store and work with strings, with each row storing a string each. The same array objects we declared earlier can now be made an array of strings:

```
char objects[][4] = {"mat", "bat", "net", "rod", "pin"};
```

The compiler automatically makes this array a 5 x 4 matrix of characters.

A few quick practice problems with integer matrices: writing functions for doing each of these tasks:

1. To check if a given matrix received as the argument is a diagonal matrix or not
2. To check if a given matrix is a lower triangular matrix or not
3. To check if a given matrix is an upper triangular matrix or not

```
int isDiagonal(int m[][COLS]) {
  int i, j;
  for (i = 0; i < ROWS; ++i)
    for (j = 0; j < COLS; ++j)
     if (i != j && m[i][j])
           return 0;   /* found a non-diagonal element that is non-zero */
  return 1;
}

int isLowerTriangular(int m[][COLS]) {
  int i,j;
  for (i = 0; i < ROWS; ++i)
    for (j = 0; j < COLS; ++j)
      if (j > i && m[i][j])
           return 0;   /* a non-zero element found in the upper triangle */
  return 1;
}

int isUpperTriangular(int m[][COLS]){
  int i,j;
  for (i = 0; i < ROWS; ++i)
    for (j = 0; j < COLS; ++j)
      if (i > j && m[i][j])
           return 0;   /* a non-zero element found in the lower triangle */
  return 1;
}
```

LABORATORY SESSION #9

Exercise 9-1: For the piece of code shown below that finds out the amount of memory allocated (in bytes) for two arrays, can you guess what gets printed?

```
int arr[] = {10,20,45,67,68};
int mat[][2] = {{10,20}, {30,40}, {50,60}};
printf("Sizeof(arr) = %lu; sizeof (mat) = %lu\n", sizeof(arr),
sizeof(mat));
```

Now when the array is passed to a function in which the size is calculated (as shown below), what do you think gets printed?

```
void foo(int arr[], int mat[][2])
{
   printf("Sizeof(arr) = %lu; sizeof (mat) = %lu\n",
                          sizeof(arr), sizeof(mat));
}
```

A copy of the program is stored at `/home/share/9.1.c`. Copy it to your working directory, compile and check the outputs. Were they what you expected? Record them in your lab notebook.

Exercise 9-2: Copy the program at `/home/share/9.2.c` into your working directory.

```
 1 #include <stdio.h>
 2 int main()
 3 {
 4   int i = 7;
 5   int arr[5] = {i, 2*i, 3*i}, *pa, *pb;
 6   pa = &arr[0];
 7   pb = arr+4;
 8   for (i = -4; i <= 0; ++i)
 9       printf("%d, ",pb[i]); /* negative array
    index! */
10   putchar('\n');
11
12   printf("pb - pa = %ld\n", pb-pa); /* pointer
    subtraction */
13   return 0;
14 }
```

Several interesting concepts are illustrated in this program. Understand these:

a. What aspects of array initialization are illustrated (lines 5, 9)? Write in your lab notebook.

b. What aspects of pointer arithmetic are illustrated (lines 9, 12)? Write them down.

c. Try subtracting two pointers of different data types (e.g., `char *` and `int *`). Try adding two pointers of the same type; multiplying; adding a floating point number to a pointer. Write down which of these the compiler permits.

Exercise 9-3: ABC bank is trying to establish ATM branches in different regions of a city. There are **`m`** branches and each branch does **`n`** transactions per day. The transaction could be a positive value indicating deposit, or negative indicating withdrawal, or zero indicating it was neither a deposit nor a withdrawal. Write a complete C program that accomplishes these tasks:

a. Reads values for **`m`** and **`n`**, and then reads and stores the transaction values in an **`m`** x **`n`** matrix. Data for the program is stored at `/home/share/9.3.txt`. Take a look at the file – the first row has values to be stored in **`m`** and **`n`**, and the rest of the matrix follows. (Tip: Use input redirection to read the data from file.)

b. Prints the branch number (same as row number) of the branch that had the maximum total deposits. Use the following function to achieve this task:

```
int computeSumOfDeposits(int row[], int size);
```

c. Prints the branch number of the branch that had the maximum transaction value (Hint: Use the `abs()` function from the `<stdlib.h>` library, or implement the equivalent functionality yourself.)

```
int computeMaxTransac(int row[], int size);
```

This function computes and returns the maximum transaction amount for the given ATM branch. You can view sample output of the program by executing `/home/share/9.3` to see how your program is expected to work.

2.3 Array of Pointers

Consider the following two declarations:

```
char arr[]="Pointers! Pointers! Pointers!"; /* strlen (arr) = 29 */
char *ptr="Pointers! Pointers! Pointers!";
```

Though both seem to be similar in terms of printing the string, i.e., `puts(arr);` and `puts(p);` will both give the same result, there are some important differences:

`arr` is a character array that has been allotted memory storage (30 bytes including the terminating NUL character) whose contents can be accessed and changed (e.g., `arr[0] = 'C';` will change the first character of the string from P to C). The identifier `arr` is also the base address of the array. Because it is a address, it can be assigned to a pointer variable, e.g., `char *q = arr;` However, being a base address, it cannot be altered, e.g., `arr = arr + 1;` or `arr = q;` are invalid expressions that throw compile-time errors.

The second declaration is that of a pointer variable named **`ptr`** (allocated 8 bytes) that stores the address of the string constant `"Pointers! Pointers! Pointers!"`. This string constant is stored somewhere in memory by the compiler, the contents of which can be read, *but which cannot be changed*. For instance, attempting to do `ptr[0] = 'C';` will result in a runtime error, because it is illegal to change a string constant. However, being a variable, the contents of ptr can be changed, e.g., `ptr = arr;` is perfectly valid.

Extending this, we can now declare an array of pointers and initialize with five names (strings) as follows:

```
char *list[] = {"Mehul", "Inzemam", "Rushikesh", "Akash", "Divyansh"};
```

`list` is an array of five elements, each of which is a pointer to a character. The five names (constant strings) are stored somewhere in memory, and their starting address is stored as successive elements of the array `list`.

To store the five names, 2D arrays can also be used, of course:

```
char names[][100] = {"Mehul", "Inzemam", "Rushikesh", "Akash",
                                                    "Divyansh"};
```

Both appear similar, and can be used to print the names by running a loop:

```
for (i = 0; i < 5; ++i)
      puts(list[i]);    /* printing each string being pointed to */
```

```
for (i = 0; i < 5; ++i)
      puts(names[i]);   /* printing each row of the 2D array */
```

But carefully note that while `list[i]` has a physical memory to store an address, `names[i]` is not a physical memory and hence can't be used as a pointer variable. The next subsection will illustrate the differences among them.

2.3.1 Sorting algorithms

Bubble sort works this way to sort n elements of an array:

```
for index i = 0 to i < n-1
  for index j = 0 to j < n-i-1
      if (arr[j] > arr[j+1])
          swap the two elements arr[j] and arr[j+1]
```

Selection sort works this way:

```
for index i = 0 to i < n-1
  for index j = i+1 to j < n
      if (arr[i] > arr[j]) /* change > to < to sort in descending order */
          swap the two elements arr[j] and arr[j+1]
```

Let us now employ selection sort to sort the names using 2D array and the array of pointers.

```
int i, j, n = 5; / * sorting 5 names */
char temp[100];
for (i = 0; i < n-1; ++i)
  for (j = i+1; j < n; ++j)
     if (strcmp(names[i],names[j]) > 0) /* an earlier name is alphabetically
                                           greater than a later name */
      {
       strcpy(temp,names[i]);       /* copy the earlier name to array temp[] */
       strcpy(names[i],names[j]);   /* copy later name into earlier location */
       strcpy(names[j],temp);       /* copy temp[] contents to later location */
      }
```

When doing the same sort using the array of pointers, we will not be able to move the strings from one location to another, for each of them is a string constant whose contents cannot be changed. Hence, instead of swapping the strings, we will swap the pointers that point to them:

```
int i, j, n = 5; / * sorting 5 names */
```

```
char *tmp;        /* it is a pointer, not an array */
for (i = 0; i < n-1; ++i)
  for (j = i+1; j < n; ++j)
     if (strcmp(list[i],list[j]) > 0) /* string pointed to by list[i] is greater
                                         than the string pointed to by list[j] */
      {           /* swap the addresses, not the strings themselves */
       tmp = list[i];
       list[i] = list[j];
       list[j] = tmp;
      }
```

The two schemes of storing strings - a 2D array or an array of pointers have their own strengths and disadvantages:

1. From the point of view of memory, the array of pointers saves space because the amount of memory allocated for each string is exactly as many characters as it contains (with one extra byte for storing the terminating \0). The 2D array wastes space, for once the length of each row is specified (column size), it cannot be changed later.

2. A 2D array, however, gives the flexibility of the user being able to modify the locations where strings are stored. This means, the strings (names) can be taken from the user when the program runs, and then stored into the 2D array:

```
int nos, i;
printf("Enter how many names you want to enter: ");
scanf("%d" , &nos);
for (i = 0; i < nos; ++i)
{
   scanf("%[^\n]", names[i]); /* passing the row address */
   getchar();     /* remove the extra \n from the buffer */
}
```

   An attempt to do such a thing using an array of pointers would result in a runtime error, as this segment shows:

```
printf("How many names do you want to enter? ");
scanf("%d" , &nos);
getchar();
char *list[nos];  /* declaring an array of nos pointers */
for (i = 0; i < nos; ++i)
{
  scanf("%[^\n]",list[i]);    /* <<<<<< HERE IS THE PROBLEM */
      getchar();
}
```

Where can a string be stored when no memory has been allocated for it? That's exactly what is being attempted here, and hence the program terminates abruptly.

What we then need is a method to be able to allocate memory for a string after taking it from the user, and then store the starting address of each of the strings thus taken in the array of pointers.

2.4 Dynamic Memory Management

When memory has to be allocated at run time by the user, such memory is allocated by the system from what is known as "heap". Some features of heap/dynamic memory allocation and management:

- The user (program) has to request for space using the functions **`malloc()`**, **`calloc()`** or **`realloc()`**. Here are their prototypes:

```
#include <stdlib.h>
void *malloc(size_t num); /* num bytes allocated  */
void *calloc(size_t nmemb, size_t num); /* an array of nmemb
                        elements, each of size num is allocated */
void *realloc(void *ptr, size_t num);
```

- The chunk of bytes allocated using these functions can only be accessed through the starting address (pointer) they return (the `void *` indicates they return a generic pointer). In other words, objects allocated in the heap memory do not have identifiers to access them.

- It is the user's responsibility to surrender the heap space after use. The `free()` function is used for this purpose.

```
void free(void *ptr);
```

- Heap memory has scope and lifetime till it is freed (or the program exists). This means, *even when allocated within a function, other functions can access the same memory as long as the starting address is made available to them*.

LABORATORY SESSION #10

Exercise 10-1: Declare an array of m pointers, each pointing to a dynamically allocated array of n integers (m and n are both user inputs). Then, take m x n inputs for the matrix elements. Next, display the entire matrix on screen in a neatly formatted manner. If you wish, you can use the data file `/home/share/10.1.txt` for taking inputs (first line contains m and n, and the remaining lines have the values for the m x n matrix).

Exercise 10-2: This program involves taking input for *m* rows of a matrix, but each row having a varying number of integer elements. Take a look at `/home/share/10.2.txt` (and copy to your current directory) to get an idea of this matrix with varying number of columns for each row. The first row contains the number of rows. The first element of each row contains the number of columns of that particular row, followed by the actual elements themselves.

a. Now declare a suitable array of pointers, dynamically allocate memory for each row, read the elements (including the number of elements as the first entry of each row), and then display the entire matrix on the screen. Verify to make sure that your output is the same as the matrix entries given in the data file you copied (`/home/share/10.2.txt`).

b. Take a look at the ATM branches problem you worked on during last week's lab (page #13 of this document). Copy into your current working directory the C program you wrote last week for this question and rename the file as `10.2.c`. Now consider the following modification of the problem statement:

   It was observed that there were few ATM branches that had very few transactions, but others that had several. The file available at `/home/share/10.2.txt` has the data – the first row has the number of branches, while the first number in each of the other rows (representing a branch) indicate the number of transactions recorded for that branch, followed by the actual transaction values.

   Now modify the code in `10.2.c` in a way the new data can be captured – using a dynamically allocated array rather than an *m* x *n* matrix allocation you had done earlier.

Exercise 10-3: Two words/phrases are said to be anagrams if one of them can be rearranged to give the other. Write a function **`isAnagram()`** that takes two strings as arguments and tells whether the first string is an anagram of the second or not. For example, the function should return a value of 1 for pair of string inputs such as "great" and "grate"; "Listen" and "siLent"; "rail safety" and "fairy tales". It returns a 0 for string inputs such as: "night" and "knight"; "great" and "Grate"; "Madam Curie" and "Radium came".

[Hint: One of the many strategies that can be used to solve this problem involves sorting the individual characters of the string! If you need to use a sorting function, you may use the code for selection sort available at `/home/share/10.3.c` in your program.]

Exercise 10-4: In the lecture class, we discussed selection sort technique that sorts an array of elements. Insertion sort is another technique for sorting whose algorithm is given below:

```
1.    Read N, Arr
2.    FOR i = 1 to N-1
      Key = Arr[i]
      j = i - 1
      WHILE ( j >= 0 and Arr[j] > Key)
            Arr[j+1] = Arr[j]
            j = j - 1
      Arr[j+1] = Key
3.    Output Arr
4.    Stop
```

a. Understand the algorithm by taking a sample array of elements {8,5,3,9,4}. Write down in your lab notebook the stepwise results of applying the above algorithm on this input.

b. Translate this algorithm into a C function which has the following prototype:

```
void insertionSort(int arr[], int num);
```

Test the function by calling it in main() with sample data.

3. STRUCTURES AND UNIONS

3.1 Structures

A structure in C is a user-defined type, with one or more fields in it. There are two ways of defining a structure:

(i) Using a "tag" for the structure:

```
struct fraction {
int numer;
int denom;
};
```

(ii) using type definition:

```
typedef struct {
int x;
int y;
} POINT;
```

Either of these ways of defining a structure does not allocate memory for the type - but merely states the type.

Now that a type is defined, variables of the type can be declared:

```
POINT p1, p2;       /* now memory is allocated for p1 and p2 */
struct fraction f1, f2, fracs[100];
```

While the `f1` and `f2` are single structures of type `fraction`, `fracs[100]` is an array of 100 `fractions`. You will learn more about array of structures in lectures this week.

- Usually, the structure definition is done outside the `main()`, globally, so that all functions are able to use the definition.
- The amount of memory allocated for a structure variable is at least the sum of number of bytes that each field occupies (the extra bytes may be due to the "padding" done for each structure.
- A structure can be initialized when declaring the variable, or later, using the . (dot) operator (member of operator).
- A structure can be assigned to another variable of the same type. Checking for the equality of 2 structures or any other operation cannot be performed.

LABORATORY SESSION #11

Exercise 11-1: In the lecture class, we discussed an example of a student structure that stores the different attributes of a student viz., name, ID number, age and CGPA.

```
struct student {
    char name[20];  // stores only the first name
    char id[14];
    int age;
    float cgpa;
} s1;
```

a. Define the structure in your program globally before the `main()` function, i.e., write the above few lines before `main()`.

b. Declare two other variables of type `struct student` having the names `s2` and `s3`, both local to `main()`.

c. Inside `main()` find out how many bytes are allocated for `s1`.

d. Accept user input for `s1` and `s2`, using the dot operator for every member. For instance, to accept the name, you would write: `scanf("%s", s1.name);` And for scanning the CGPA, you would type: `scanf("%f", &s1.cgpa);` and so on.

e. Assign the contents of `s1` to `s3` in one stroke.

f. Now check if `s2` is equal to `s3`, and print a message appropriately.

g. Include the following function definition in your program:

```
void changeData(struct student s) {     // receives a structure argument
      strcpy(s.name,"Madhav");
      s.age = 20;
      s.cgpa = 7.33;
}
```

Call this function from `main()` as follows:

```
changeData(s1);   // takes a structure as argument
```

Print the contents of `s1` in `main()` before and after calling `changeData()`. What can you conclude about the mechanism of passing structures to functions?

h. Print out the address of `s3`. Next, print out the addresses of the individual elements of `s3`. Note that the "member of operator" has the highest precedence

among all operators. In other words, `&s1.age` will mean `&(s1.age)`, even without the parenthesis.

i. How would you declare a variable to store the address of `s1`?

Exercise 11-2: Modify the function you wrote for question 10-3 (previous lab) so that a new function called **`isAnagram2()`** ignores the case of the letters of the strings that are compared. For example, this function would return a value 1 even for the two strings "Madam Curie" and "Radium came". [Hint: You may use `toupper()` or `tolower()` whose declarations are available in `<ctype.h>` header file.]

A third version of this function, **`isAnagram3()`** that is not only case-insensitive, but also disregards any spaces found in the string has been implemented below, incompletely. (Note: A copy of this function, including `main()`, is available at: `/home/share/11.2.c` for you to copy into your directory.) For instance, this function is supposed to recognize each of the pairs such as "dormitory" and "dirty room", "Slot Machines" and "Cash lost in me" as anagrams. The logic is to add up the ASCII values of each character of the strings (excluding space and tab) and compare the sums. If the sums are equal and so are the number of non-space characters of both strings, the strings are anagrams, else they are not!

```
int isAnagram3(char *w1, char *w2) {
  int i, sum_w1 = 0, sum_w2 = 0, no_chars_w1 = 0, no_chars_w2 = 0;
  for (i = 0; w1[i] ; ++i)
  {
    if (isspace(w1[i]))
          /* >>>>> a statement s1 goes here <<<<< */
    no_chars_w1++;
    sum_w1 += /* >>>>> an expression e1 goes here <<<<< */ ;
  }
  for (i = 0; w2[i] ; ++i)
  {
    if (isspace(w2[i]))
          /* >>>>> s2 goes here <<<<< */
    no_chars_w2++;
    sum_w2 += /* >>>>> e2 goes here <<<<< */ ;
  }
  if ( /* >>> e3 here <<< */ && /* >>> e4 here <<< */)
    return 1;
  else
    return 0;
}
```

a. Complete the function and run the program.
b. Does this logic work for all cases, or does it fail in any? Can you think of an example where it fails?
c. (Homework) Can you design another algorithm for finding if two strings are anagrams? Hint: Think of how you would solve using pen and paper, and convert that into a C program.

3.2 Unions

A union is very similar to a structure - with members in it, but with one major difference: each member is not allocated space, but only the largest of them is. Imagine you have to withdraw money for your travel from Pilani to Delhi that you can go by cab (costing Rs. 3400), train (Rs. 500) or bus (Rs. 200). If you are not sure which mode you will use, but have to withdraw the money from ATM, which amount would you go for? The largest of the three possibilities, right? Now consider this union definition:

(i) Using a "tag" for the union:

```
union number {
int natural;
float decimal;
};
```

(ii) using type definition:

```
typedef union {
int cab_fare;
float train_fare;
short bus_charge;
} FARE;
```

Either of these ways of defining a union can be used, but neither allocates memory for the union type - but merely defines the type. Declaring members of the type allocates memory, as shown here:

```
union number n;    /* sizeof (float), i.e., 4 bytes allocated for n */
FARE travel;/* sizeof (float), i.e., 4 bytes allocated for money */
```

[Note that had these declarations been of `struct`, then the memory allocated for each structure would have been at least the sum of the sizes of the individual member elements (i.e., 4 + 4 = 8 bytes for `union number`, and 4 + 4 + 2 = 10 bytes for `FARE`).]

All other operations on unions are same as that of structure (you can assign values, you can assign one union to another, but can't do much else).

The user will decide which of the fields (member elements) of the union are to be used at any given time. Take another example - you have a box that can store either a book, or a pen or an eraser. Naturally, the size of the box should be according to the largest item you would like it to hold at a given time - which would be that of the book. But, at any given instance, you can store only one of the three - either a book or a pen or an eraser. For instance,

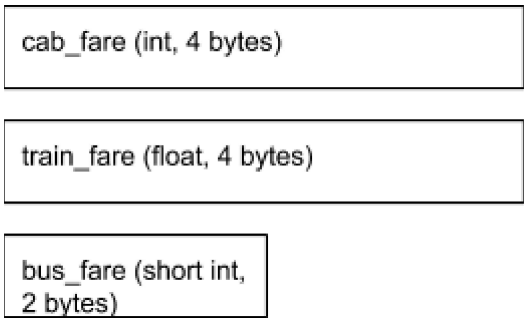
```
travel.bus_charge = 190;
printf("%hd\n", travel.bus_charge);
```

This would print the number 190 on the standard output.
Later on, you can change the value and print accordingly:

```
travel.train_fare = 490.25;
printf("%f\n", travel.train_fare);
```

Internally, this would store the floating point value starting at the same memory location as what was stored earlier as a `short int`.

The fields of the union all begin at the same memory location, as shown here:



In fact, a better way of conceptualizing this would be imagine all the three boxes superimposed on each other, left-aligned.

Now, if the user does something like this:

```
1     FARE travel;
2     travel.bus_charge = 190;
3     printf("Bus charge = %hd\n", travel.bus_charge);
4     travel.train_fare = 490.25;
5     printf("Train fare = %f\n", travel.train_fare);
            /* so far, so good! */
6     printf("Bus charge = %hd\n", travel.bus_charge);      /* weird! */
```

The output would be:

```
Bus charge = 190
Train fare = 490.250000
Bus charge = 8192
```

The first two lines of output are nothing unusual. But since in line 4, the bit pattern at the location (4 bytes) has been modified to store the IEEE equivalent of 490.25, when you attempt to print in line 6 the value at the first two bytes (`short int`) of the four bytes, you get the converted equivalent of the bit pattern at the first two bytes.

Lesson to be learned: it is the programmer's responsibility to carefully retrieve the data from a union variable.

Example

"A bit pattern stored in computer memory is interpreted differently depending on the context. Consider the 32 bits abbreviated by the hexadecimal notation 0xC0000000.

a. If this 32-bit entity represented a signed integer (in 2's complement form), what value would it represent? Show all steps how you arrived at the answer. (Note: you may write the final answers in powers of 2.)

b. If these 32 bits represented a floating point number (that used the IEEE-754 format), what value is stored at the memory location? Show all steps."

Let us try to use the concept of union to find the answers.

```
union int_or_float {
      int i;
      float f;
      } labq;
labq.i = 0xC0000000;
printf("%d\n", labq.i); /* Printing the bit pattern as integer */
printf("%f\n", labq.f); /* Printing the same pattern as a float */
```

And the output produced after compiling and running this piece of code is:

```
-1073741824
-2.000000
```

Hence, the pattern represents -2^{30} when interpreted as an integer, and **-2.0** when interpreted as a floating-point number.