C Programming

Lesson 5: Pointers

Table Of Contents

Lesson Objectives

- **In this lesson, you will learn about:**
  - The method in which pointer arithmetic and pointer is used with functions
  - The method in which the pointers are used to process one-dimensional and two-dimensional arrays

Lesson Coverage

- **In this lesson, you will learn the following topics:**
  - Pointers and Functions:
    - Call by Value, Call by Reference
    - Pointers to Functions: Declaration, Invoking function using Pointer
    - Functions returning Pointers
  - Pointer and Array:
    - Pointers and multidimensional (Two dimensional) arrays
    - Array of pointers (int, string)

Lesson Coverage

- **In this lesson, you will learn the following topics (contd.):**
  - Pointers to Pointers
  - Dynamic Memory Allocation:
    - List of Pointer Declarations examples
    - Command Line Arguments

Concept of Pointers and Functions

- **A function can take a pointer to any data type, as argument and can return a pointer to any data type**
  - For example, consider the following function definition.

```
double *maxp(double *xp, double *yp)
{
  return (*xp >= *yp ? xp : yp);
}
```

  - It specifies that the function maxp() returns a pointer to a double variable, and expects two arguments, both of which are pointers to double variables.

Concept of Pointers and Functions

- **The function de-references the two argument pointers to get the values of the corresponding variables, and returns the pointer to the variable that has the larger of the two values .**

- **Thus given that, double xp=1.5, yp=2.5, *mp;**
  the statement mp = maxp(&xp, &yp);
  makes mp point to v.

Concept of Pointers to Functions

- **Functions have addresses just like data items.**
- **A pointer to a function can be defined as the address of the code executed when the function is called.**
- **A function's address is the starting address of the machine language code of the function stored in the memory.**
- **Pointers to functions are used in the following tasks:**
  - writing memory resident programs
  - writing viruses or vaccines to remove the viruses

Pointer to Functions – Declaration

- **The declaration of a Pointer to a Function requires the function's return type and the function's argument list to be specified along with the pointer variable.**

- **The general syntax for declaring a pointer to a function is:**

```
return-type (*pointer variable)(argument_list);
```

- **Thus, the declaration int (*fp)(int i, int j);**

  declares fp to be a variable of type "pointer to a function that takes two integer arguments and returns an integer as its value".

Pointer to Functions – Example

- **Example: Let us see an example for Pointer to Functions:**

```
#include<stdio.h>
int stud_detail(int rollno)
{
        return rollno;
}
float mark_detail(float mark)
{
        return mark;
}
```

Pointer to Functions – Example

- **Example contd. :**

```
void main()
{
        int (*s_ptr)(int); /*declaring pointer to function*/
        float (*m_ptr)(float); /*declaring pointer to function*/
        int rollno=5;
        float mark=1.5;
        s_ptr=stud_detail;
        m_ptr=mark_detail;
        printf("RollNo=%d Mark=%f\n",s_ptr(rollno),m_ptr(mark));
}
```

Demo on Invoking Functions

- **Pointer_to_fun.c and pointer_to_function.c demo**

Invoking Functions - Example

- **Example: Let us see an example on Invoking Functions:**

```
/* Example : Invoking function using pointers */
# include <stdio.h>
void main(void)
{
  unsigned int fact(int);
  unsigned int ft,(*ptr)(int);
  int num;
  ptr=fact;     /* assigning address of fact() to ptr */

  printf("Enter integer whosefactorial is to be found:");
  scanf("%d",&num);
  ft=ptr(num);    /* call to function fact using pointer ptr */
  printf("Factorial of %d is %u \n",num,ft);
}
```

Invoking Functions – Example

- **Example contd.:**

```
unsigned int fact(int num)
{
  unsigned int index,ans;
  if (num == 0)
      return(1);
  else
  {
     for(index=num, ans=1; index>1 ; ans *= index--);
          return(ans);
  }
}
Output:
          Enter integer whose factorial is to be found : 8
            Factorial of 8 is 40320
```

Concept of Functions returning Pointers

- **A function can return a pointer like any other data type.**
- **However, it has to be explicitly mentioned in the calling function and in the function declaration.**
- **While retaining pointers, return the pointer to global variables or static or dynamically allocated address.**
- **Do not return any addresses of local variables because it does not exist after the function call.**

Functions returning Pointers - Example

- **Example:**

```
/* Example: Function returning pointers */
/* Program to accept two numbers and find
 greater number */
# include <stdio.h>
void main(void)
{
        int a,b,*c;
        int* check(int*,int*);
        printf("Enter two numbers : ");
        scanf("%d%d",&a,&b);
        c=check(&a,&b);
        printf("\n Greater numbers : %d",*c);
}
```

check function takes two integers pointers as arguments and returns a pointer to integer

Functions returning Pointers - Example

- **Example (contd.):**

```
int* check(int *p,int *q)
{
        if(*p >= *q)
                return(p);
        else
                return(q);
}
```

Concept of Pointers and Arrays

- **Arrays are internally stored as pointers.**
- **A pointer can efficiently access the elements of an array.**
  - Example:

```
#include <stdio.h>
void main(void)
{
  int ar[5]={10, 20, 30, 40, 50};
  int i, *ptr;
  ptr = &ar[0]; /* same as ptr = ar */
  for(i=0; i<5; i++)
  {
     printf("%d-%d\n", ptr, *ptr);
     ptr++;
  }
```

increments the pointer to point to the next element and not to the next memory location

Concept of Pointers and Arrays

- Example (contd.):

```
}
Output:
        5000-10
        5004-20
        5008-30
        5012-40
        5016-50
```

Concept of Pointers and Arrays

- **ar ≡ &ar[0] ≡ 5000**
  - Hence, *ar ≡ *(&ar[0]) i.e. *ar ≡ ar[0] or *(ar+0) ≡ ar[0]
  - To make the above statement more general, we can write:
    - *(ar+i) ≡ ar[i];
      - where, i=0,1,2,3,...

	5000	5004	5008	5012	5016
ar	10	20	30	40	50
	ar[0]	ar[1]	ar[2]	ar[3]	ar[4]

Concept of Pointers and Arrays

- Hence any array element can be accessed using pointer notation, and vice versa. It will be clear from following table:

char c[10], int i;	
Array Notation	Pointer Notation
&c[0]	c
c[i]	*(c+i)
&c[i]	c+i

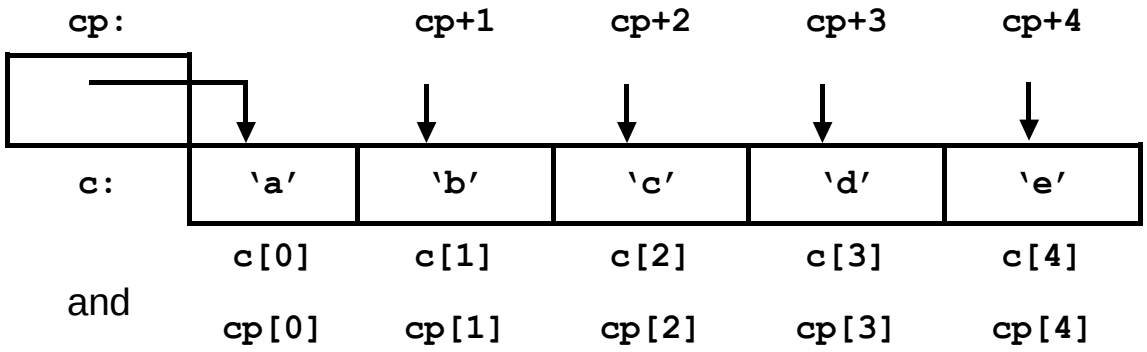
Concept of Pointers and Arrays

For example, given that

```
char c[5] =  {'a', 'b', 'c', 'd', 'e'} ;
  char *cp ;
```

and

```
cp = c;
```



and

	cp[0]	cp[1]	cp[2]	cp[3]	cp[4]
c[0]	↛ 'a'		*cp	↛ cp[0]
c[1]	↛ 'b'		*(cp+1)	↛ cp[1]
c[2]	↛ 'c'		*(cp+2)	↛ cp[2]
c[3]	↛ 'd'		*(cp+3)	↛ cp[3]
c[4]	↛ 'e'		*(cp+4)	↛ cp[4]

Concept of Pointers and Arrays

- **Using this concept, we can write the program as shown below:**

```
# include <stdio.h>
void main(void)
{
int ar[5]={10,20,30,40,50};
int i;
for(i=0;i<5;i++)
  printf("%d-%d\n",(ar+i),*(ar+i));
}
```

Concept of Pointers and Arrays

- **C does not allow to assign an address to an array.**

- **For example:**
  - Consider the array: ar=&a;
  - It is invalid.

- **The main difference between an array and a pointer is that an array name is a constant, (a constant pointer to be more specific), whereas a pointer is a variable.**

Concept of Pointers & Multidimensional Arrays

- **A Multi-dimensional array in C is really a one-dimensional array, whose elements are themselves arrays, and is stored such that the last subscript varies most rapidly. The name of the multi-dimensional array is, therefore, a pointer to the first array. Thus, the following declaration holds true:**
  - int matrix[3][5];

Concept of Pointers & Multidimensional Arrays

- **For example: The element matrix[i][j] can be referenced using the pointer expression *(*(matrix+i)+j)**
  - Matrix is a pointer to the first row;
  - matrix + i is a pointer to the ith row;
  - *(matrix + i) is the ith row which is converted into a pointer to the first element in the ith row;

Concept of Pointers & Multidimensional Arrays

- **A two-dimensional array is a collection of one-dimensional array as a pointer to a group of contiguous one-dimensional arrays.**

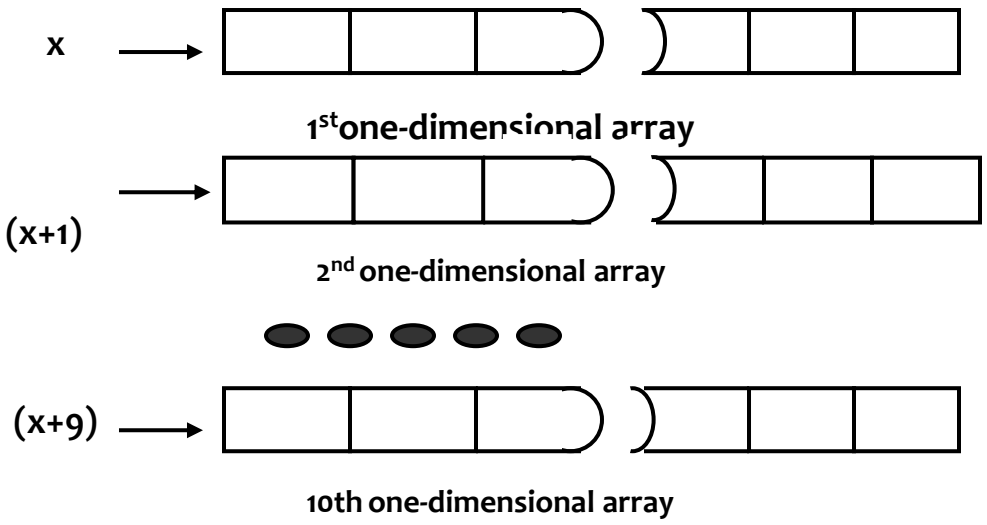
- **So a two-dimensional array declaration can be written as the following data-type**
  - (*array)[expression 2];
    - where,
      - data-type refers to the data type of the array,
      - *array is the pointer to array and
      - expression2 indicate the max. number of elements in a row

- **Note: The parentheses that surround the array name must be present.**

Concept of Pointers & Multidimensional Arrays

- **Example:**
  - Suppose x is a two-dimensional integer array having 10 rows and 20 columns, then it can be declared as follows:
    - int (*x)[20];
      - Here, x is defined to be a pointer to a group of contiguous, one-dimensional, 20-element integer arrays.
      - Thus, x points to the first 20-element array, which is actually the first row (row 0) of the original two-dimensional array.
      - Similarly, (x + 1) points to the second 20-element array, which is the second row (row 1) of the original two-dimensional array, and so on.

Concept of Pointers & Multidimensional Arrays



A 2-dimensional array in memory

Example: Suppose that x is a two-dimensional integer array having 10 rows and 20 columns, as declared in the previous example.

Then the item in row 2, column 5 can be accessed by writing either
x[2][5] or *(*(x + 2) + 5)

Concept of Array of Pointers

- **An array of pointers is collection of addresses.**

- **Addresses can be address of "isolated variables" or addresses of "array elements" or any other addresses.**
  - For example: Consider the following declaration.

```
char *day[7];
```

  - It defines day to be an array consisting of seven character pointers.

Concept of Array of Pointers

- **The elements of a pointer array, can be assigned values by following the array definition with a list of comma-separated initializers enclosed in braces.**
  - For example: Consider the following declaration.

```
char *days[7] = {“Sunday”, “Monday”, “Tuesday”, “Wednesday”,
“Thursday”, “Friday”, “Saturday”}
```

  - Here, the necessary storage is allocated for the individual strings, and pointers to them are stored in array elements.

Array of Pointers - Example

- **ar contains addresses of isolated integer variables m, n, o, and p.**
- **The for loop in the program picks up addresses present in ar, and prints the values present at these addresses.**

```
/* Example :Array of Pointers */
# include <stdio.h>
void main(void)
{
   int m = 25, n = 50, o = 60, p = 74;
   int i,*ar[4]={&m,&n,&o,&p}
   for(i=0; i<4; i++)
      printf("%d\t",*(ar+i));
}
```

Array of Pointers - Example

- **Memory representation is shown below:**

int variables	m	n	o	p
values stored in variables	25	50	60	74
addresses of variables	4002	5013	3056	9860

array of pointers	ar[0]	ar[1]	ar[2]	ar[3]
Elements of an array of pointers	4002	5013	3056	9860

- **An array of pointers can contain addresses of other arrays.**

Concept

- **A multidimensional array can be expressed in terms of an array of pointers.**
- **A two-dimensional array can be defined as a one-dimensional array of pointers by the following expression:**

```
data-type *array[expression 1];
```

Concept

- **Note that the array name and its preceding asterisk are not enclosed in parentheses in this declaration.**
- **Thus, a right-to-left rule first associates the pairs of square brackets with array, defining the named data item as an array.**
- **The preceding asterisk then establishes that the array will contain pointers.**

Concept

- **Example:**
  - Suppose that x is a two-dimensional char array having 10 rows and 20 columns, then it can be defined as:

```
char *x[10];
```

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

  - Note that the number of elements within each row is not explicitly specified.

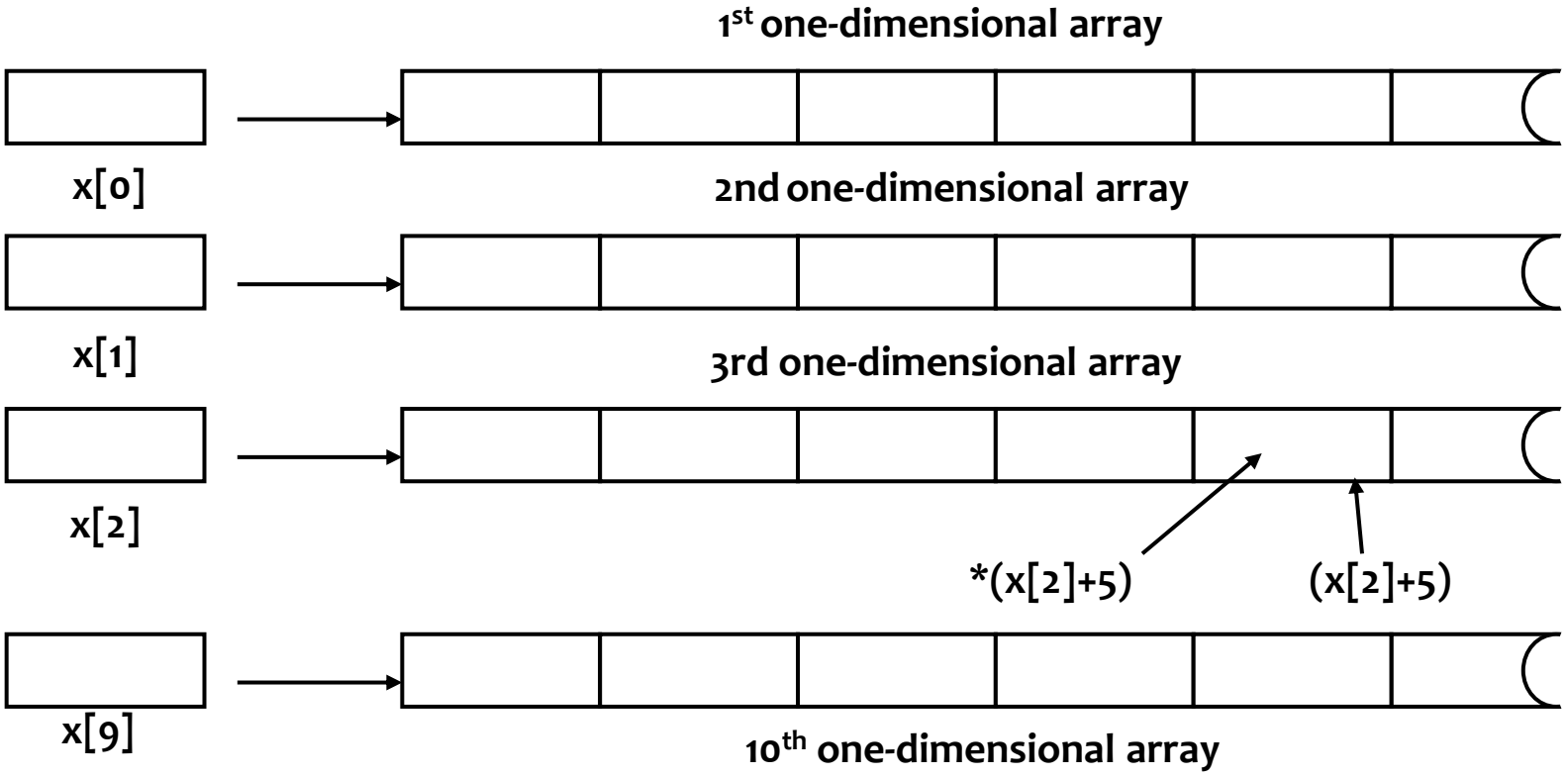
Concept

- In individual array element, such as x [2] [5], can be accessed by writing the following:

```
*(x[2]+5);
```

- In this expression, x[2] is a pointer to the first element in row 3, so that (x[2]+5) points to index 5 (actually, the sixth element) within row 3. The data item of this pointer, *(x[2]+5), therefore refers to x[2][5].

Concept



- Here x is an array of 10 pointers (x[0] to x[9]).
- Memory from the heap has to be allocated for each pointer so that valid data can be stored in it.

Concept of Pointer to Pointer

- **The data item pointed to by a pointer can be an address of another data item.**
- **Thus a given pointer can be a pointer to a pointer to a data item.**
- **Accessing this data item from the given pointer, requires two levels of indirection:**
  - The given pointer is dereferenced to get the pointer to the given data item.
  - The later pointer is dereferenced to get to the data item.

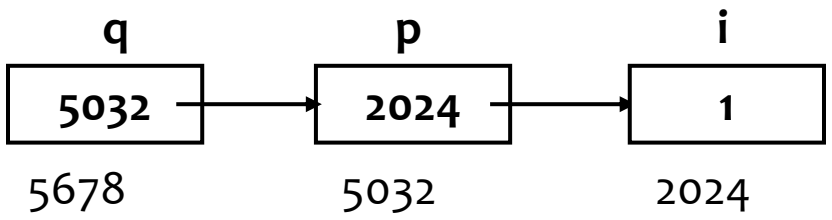
Concept of Pointer to Pointer

- **The general format for declaring a pointer to pointer is:**
  - data-type **ptr_to_ptr;

- **Here, the variable ptr_to_ptr is a pointer to a pointer pointing to a data item of the type data_type.**
  - For example: Consider the following declarations:
    - int i=1; int *p,**q;
    - These declare i as an integer, and p as a pointer to an integer.
    - The address can be assigned as follows:
      - p=&i; /* p points to i */
      - q=&p; /* q points to p */

Concept of Pointer to Pointer

- **The relationship between i, p and q is pictorially depicted as shown:**



  - This implies that q is the pointer to p, which in turn points to the integer i.

- **Consider the following expression:**
  - **q;
    - It means "apply the dereferencing operator to q twice"
    - The variable q is declared as :
      - int **q;

Concept of Pointer to Pointer

- **To get the value of i, starting from q, we go through two levels of indirection. The value of *q is the content of p which is the address of i, and the value of **q is *(&i) which is 1.**

- **It can be written in different ways as follows:**
  - *(*q) = i =1
  - *(p) = i =1
  - *(*(&p)) = i =1
  - **(q) = i =1

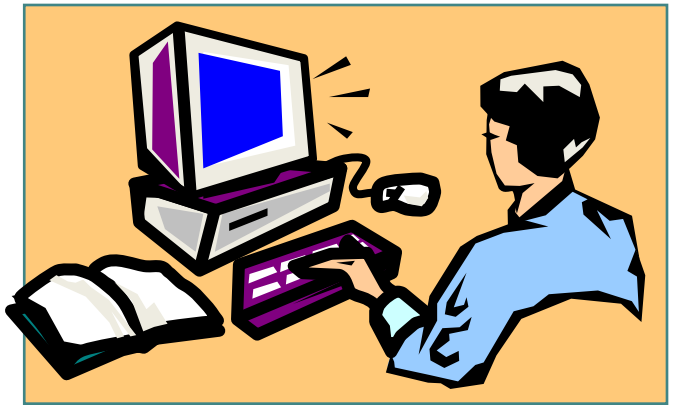
Concept of Pointer to Pointer

- **Thus each of the expressions has the value 2:**
  - i+1;
  - *p+1;
  - **q+1;

- **There is no limit on the number of levels of indirection, and a declaration such as given below is possible:**
  - int ***p;

- **Thus, ***p is an integer.**
- ****p is an pointer to an integer.**

Demo on Pointer to Pointer



Sample Code for Pointer to Pointer

- **Pointers to pointers offer flexibility in handling arrays, passing pointer variables to functions, and so on.**

```
/* Example :Pointers to Pointers */
#include <stdio.h>
void main(void)
{
   int data;
   int *iptr;     /* pointer to an integer data */
   int **ptriptr; /* pointer to int pointer */
   iptr = &data;  /* iptr points to data */
   ptriptr = &iptr; /* ptriptr points to iptr */
   *iptr = 100;     /* same as data=100 */
```

Sample Code for Pointer to Pointer

```
    printf("Variable data :%d \n", data);
    **ptriptr = 200; /* same as data=200 */
    printf("variable data :%d \n", data);
    data = 300;
    printf("ptriptr is pointing to :%d \n", **ptriptr);
}
```

Concept of Dynamic Memory Allocation

- **During Dynamic Memory Allocation, the memory is allocated and de-allocated at runtime.**
- **C provides a collection of dynamic memory management functions.**
- **Their prototypes are declared in alloc.h header file (under Borland C) and in malloc.h header file (under Unix and Windows).**

Concept of Dynamic Memory Allocation

- **Allocation Functions:**

Function Prototype	Description
void* malloc(size);	Allocates memory of given size in terms of bytes and returns the address of the first location
void *calloc(nitems, size)	Similar to malloc(), except it needs two arguments, that is, the no. of items to be allocated & the size of each item
void *realloc(void *block, size);	Resizes the block, which is already allocated according to the given size

Concept of Dynamic Memory Allocation

- **malloc():**
  - Suppose that x is to be defined as a one-dimensional, 10-element array of integers. Then it is possible to define x as a pointer variable rather than as an array. Thus we can write:

```
int *x;
instead of
int x[10];
or instead of
# define SIZE 10
int x[SIZE];
```

  - However, x is not automatically assigned a memory block when it is defined as a pointer variable. Whereas block of memory large enough to store 10 integer quantities will be reserved in advance when x is defined as an array

Malloc-Example

```
#include<stdio.h>
#include<stdlib.h>
#define Null 0
main()
{
        char *buffer;
        /*allocating memory*/
        if((buffer = (char*)malloc(10))==NULL)
```

Malloc-Example

```
{
   printf("malloc failed");
   exit(1);
}
printf("buffer of size %d created \n",_msize(buffer));
strcpy(buffer,"hyderabad");
printf("\n buffer contains : %s \n",buffer);
/*reallocation*/
if((buffer = (char*)realloc(buffer,15))==NULL)
```

Malloc-Example

```
{
      printf("realloc failed");
        exit(1);
}
printf("buffer  size modified \n");
strcpy(buffer,"secunderbad");
printf("\n buffer now contains : %s \n",buffer);
/*freeing memory*/
free(buffer);}
```

Concept of Dynamic Memory Allocation

- **calloc():**
  - The calloc() function works exactly similar to malloc(). The exception is that it needs two arguments as against the one argument required by malloc().
  - General format for memory allocation using calloc() is

```
void *calloc(nitems,size);
```

    - where,
      - nitems: The number of items to allocate
      - size: Size of each item

Concept of Dynamic Memory Allocation

- **calloc() (contd.):**
  - For example:

```
ar=(int *)calloc(10,sizeof(int));
```

    - The above allocates the storage to hold an array of 10 integers and assigns the pointer to this storage to ar. While allocating memory using calloc(), the number of items which are allocated, are initialized.

Difference between malloc() and calloc()

- **There is one major difference and one minor difference between the two functions.**
  - The major difference is that malloc() does not initialize the allocated memory. The first time malloc() gives you a particular chunk of memory, the memory might be full of zeros.
  - calloc() fills the allocated memory with all zero bits. That means that anything that you are going to use as a char or an int of any length, signed or unsigned, is guaranteed to be zero. Anything you are going to use as a pointer is set to all zero bits.
  - The minor difference between the two is that calloc() returns an array of objects; malloc() returns one object. Some people use calloc() to make clear that they want an array.

Concept of Dynamic Memory Allocation

- **realloc():**
  - General format for memory allocation using realloc() is as follows:

```
void *realloc(void *block,size);
```

    - where,
      - Block: Points to a memory block previously obtained by calling malloc(), calloc(), or realloc()
      - Size: New size for allocated block.

Concept of Dynamic Memory Allocation

- **realloc() (contd.):**
  - For example:
    - Suppose the following is true.

```
char *cp;
cp=(char *)malloc(sizeof("computer");
strcpy(cp,"computer");
cp=(char *)realloc(cp,sizeof("compute");
```

    - Then cp points to an array of 9 characters containing string "computer".

Concept of Dynamic Memory Allocation

- **realloc() (contd.):**
  - For example (contd.):
    - The following function call, discards the trailing '\0' and makes cp point to an array of 8 characters containing the string "compute".

```
cp=(char *)realloc(cp,sizeof("compute");
```

    - Whereas the following call, makes cp point to an array of 16 characters.

```
cp=(char *)realloc(cp,sizeof("computerization");
```

Concept of Dynamic Memory Allocation

- **De-allocation Function:**
  - The function prototype is as follows:

```
free(ptr);
```

  - It de-allocates memory which was allocated by malloc(), calloc() and realloc(). ptr is the pointer returned by allocation function.

Concept of Command Line Arguments

- **Parameters or Values can be passed to a program from the command line which are received and processed in the main function.**
- **Since the arguments are passed from the command line, they are called as "command line arguments".**
- **They are used frequently to create command files.**
- **All commands on the Unix Operating System use this concept.**

Concept of Command Line Arguments

- **For example:**

```
C:\>CommLineTest.exe arg1 arg2 arg3…
```

  - where,
    - CommLineTest.exe: It is the executable file of the program.
    - arg1, arg2, arg3… : They are the actual parameters for the program

Concept of Command Line Arguments

- **Two built in formal parameters are used to accept parameters in main.**

```
void main( int argc, char *argv[])
```

  - where,
    - argc : contains number of command line arguments. (type-int)
    - argv : A pointer to an array of strings where each string represents a token of the arguments passed. It is a character array of pointers

Concept of Command Line Arguments

- **For example:**

```
C:\>Tokens.exe abc 10 xyz
```

  - The value of argc will be 4.
  - The contents of argv will be as follows:
    - argv[0] : “Tokens.exe”
    - argv[1] : “abc”
    - argv[2] : “10”
    - argv[3] : “xyz”

Pointers-functions

- **Avoid returning the address of local variable from function.**
  - Variables that are local to a function go on "the stack". When the function returns/ends, all the data on the stack is popped off (discarded). So if you return a pointer to it, you are returning a pointer to a variable that no longer exists (returning an address that is no longer valid) and may be overwritten / garbage.

Bad Practice

```
char *foo()
{
static temp[80] =
{NULL};
//do something
with temp
return temp;
} //returning local
variable
```

Pointers-functions

- This is not a good practice as this will give warning at compile time and error at run-time.
- If you want to return a variable from a function, you either need to make the variable static which means it doesn't get deleted once the function ends or you need to allocate memory for the variable (using malloc())

Using MALLOC

```
char*returnMessage(char*mess
age)
{
char *msg = (char*)
malloc(sizeof(char) * 100);
char msgTest[] = "Hello how are
you";
strncpy(msg, msgTest,
sizeof(msgTest));
 return msg;
}
```

Using STATIC

```
char *foo()
{
static temp[80] =
{NULL};
//do something
withtemp
return temp;
}
```

Pointers-malloc

- **malloc (0) return may return a null pointer or a pointer to 0 byte.**
  - The behavior is implementation dependent.
  - Lesson to be learnt is that your code must take care not call malloc (0) or be prepared for the possibility of a null return;

- **Be careful while the assignment of malloc :**

```
char *p
*p = malloc(10);//bad practice
 p = malloc(10);//good practice
```

Pointers-strings

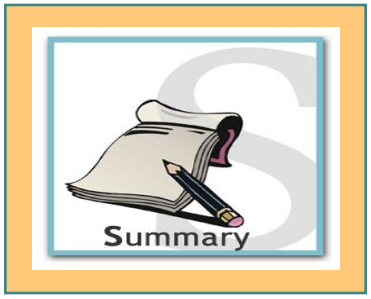
- **Prefer to use pointer strings instead array of character as string manipulation becomes easy.**

```
char str1[]="hello";
   char *s ="hello";
            str1 = 'Bye';//not allowed
  s = "bye";//allowed
```

```
char str1[]="hello";
            char str2[10];
char *s ="good morning";
char *p;
str2 = str1;//not allowed
p = s;//allowed
```

Summary

- **In this lesson, you have learnt:**
  - The address of a data item is called a "**pointer**" to the data item and a variable that holds an address is called a "**pointer variable**".
  - The address of (&) operator, when used as a prefix to the variable name, gives the address of that variable.
  - The relational comparisons <, <=, >, >= are permitted between pointers of the same type and the result depends on the relative location of the two data items pointed to.



Review Questions

- **Question 1: The data item can be accessed if we know the ___ of the first memory cell.**
- **Question 2: Null Pointer is same as un initialized pointer.**
  - True/False
- **Question 3: In pnum = &num, pnum is called as: ___.**

Review – Match the Following

1. int p(char *a[]);	1. p is a function that returns pointer to integer, argument is void
2. int (*P)(char *a) ;	2. p is pointer to integer.
3. int *p(void) ;	3. P is a pointer to a function that returns integer, argument is char pointer
4. int *p ;	4. P is a function that returns integer, accepts array of pointers to character as argument.

Lab

- **Lab 5**