C-Refresher: Session 06
Pointers and Arrays

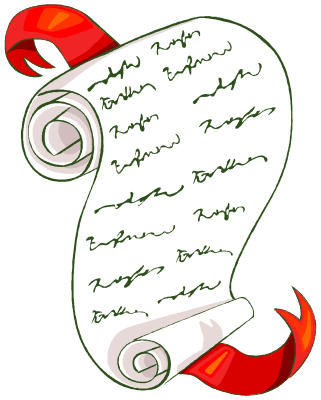
Arif Butt
Summer 2017

I am Thankful to my student Muhammad Zubair bcsf14m029@pucit.edu.pk for preparation of these slides in accordance with my video lectures at

http://www.arifbutt.me/category/c-behind-the-curtain/

Today's Agenda

- Introduction to Pointers
- The concept of NULL in C
- Pointer Arithmetic
- Pointers and const keyword
- 1D Integer Arrays and Pointers
- 2D Integer Arrays and Pointers
- 1D Character Arrays and Pointers
- 2D Character Arrays and Pointers



Introduction to Pointers

Why Pointers?

Introduction to Pointers(cont…)

❑**Reason:**

1. We can write faster and more efficient code using pointers and their arithmetic, because they are more closer to the hardware
2. They support dynamic memory allocation
3. We can protect the data that we pass to functions as pointers
4. We can pass compressed data structures to functions without incurring large overhead on the stack

- And many others too

Introduction to Pointers(cont…)

- How to declare a pointer?
- **Syntax**
  - `datatype* variable;` or `datatype *variable`
- e.g. `int i;`
- `int* x=&i;` /*ampersand(`&`) is the address of operator, it returns the address of its operand, here it returns the address of `i`*/
- To get the value of `i` using pointer, we will use the following statement
- `i=*x;` /*asterisk(`*`) symbol, here, is being used to get the value of `i`, i.e. for dereferencing `x`*/

Introduction to Pointers(cont…)

- Note we see that Asterisk(*) symbol has three meanings
  i. For declaration of a pointer
  ii. For dereferencing a pointer
  iii. As an Arithmetic operator

❑**Some basic properties of Pointers**

- Pointer type must be same as the datatype of the variable whose address it's going to hold
- Size of a pointer is same irrespective of its datatype
- A void pointer can be used for polymorphic behaviour, which can hold the address of variable of any datatype

Introduction to Pointers(cont…)

```
//program showing basic use of Pointer
#include<stdio.h>
#include<stdlib.h>
int main(){
  int i=54;
  int * ptr=&i;
  printf("value of i is %d\n",i);//value of i
  printf("value of &i is %p\n",&i);//address of i
  printf("value of ptr is %p\n",ptr);//value of ptr=add of i
  printf("value of &ptr is %p\n",&ptr);//address of ptr
  printf("value of *ptr is %d\n",*ptr);//value of i
  printf("value of sizeof(ptr) is %ld\n",sizeof(ptr));
  printf("value of sizeof(i) is %ld\n",sizeof(i));
  return 0;}
```

Introduction to Pointers(cont…)

- The Output of the above program is:

```
Value of i is 54
Value of &i is 0x7ffc5898984c
Value of ptr is 0x7ffc5898984c
Value of &ptr is 0x7ffc58989840
Value of *ptr is 54
Value of sizeof(ptr) is 8
Value of sizeof(i) is 4
```

- Here addresses are the logical addresses, when you execute the program again, it produces different logical address

Introduction to Pointers(cont…)

- Old Operating systems didn't produce different logical addresses, when you reload the program
- This same logical address generation often caused the injection of malicious codes into the program
- However, newer operating systems, like Linux, provide **address space randomization**
- You can disable this in Linux using the following command
- `$sudo sysctl -w kernel.randomize_va_space=0`
- Now, it will produce same logical addresses, no matter how many times you execute the program
- You can enable this again by setting it equal to `1`

Concept of NULL in C

- Concept of NULL in C is used in
  i. null statement => a statement having nothing but a semicolon(;)
  ii. null string => an empty string--a string containing no characters just ASCII null character
  iii. ASCII NUL => a byte containing all zeroes
    - `$asci` command can be used to check it
  iv. NULL pointer

Concept of NULL in C(cont…)

- **NULL Pointer**
  - Whenever a pointer is declared like
  - `int* ptr;` /*here `ptr` points to unknown memory location. Never dereference an uninitialized pointer*/
  - It's always better to initialize a pointer with `NULL` like
  - `int* ptr = NULL;` //this is a `NULL` pointer
    - //NULL is a macro defined in `stdio.h` that says

      ```
      #define NULL((void*)0)
      ```

Concept of NULL in C(cont…)

- A `NULL` pointer can also be written like
- `int* ptr=0;`
  - `//0` is an overloaded operand, it automatically casts to a `NULL` pointer
  - But `int a=0;` //this stores `0` in `a`
- A new concept related to `NULL` pointer is
- `if(ptr)`//it returns a binary zero `if( ptr==NULL)`
- So `if(ptr)` is equivalent to `if(ptr!=NULL)`

Example Program for NULL Pointer

```
#include<stdio.h>
#include<stdlib.h>
int main(){
  int* ptr=NULL;
  printf("value of ptr is %p\n",ptr);
  printf("value of ptr is %d\n",ptr);
  printf("value of sizeof(ptr) is %ld\n",sizeof(ptr));
  if(ptr)//if(ptr!=NULL)
    printf("ptr does not contain NULL\n");
  else
    printf("ptr contains NULL\n");
  printf("Value of *ptr id %d\n",*ptr);
  return 0;}
```

Example Program for NULL Pointer

- The output of the above program is:

```
value of ptr is (nil)
value of ptr is 0
value of sizeof(ptr) is 8
ptr contains NULL
Segmentation fault (core dumped)
```

Pointer Arithmetic

- Operators which can be used with pointers are
  - i. `*, &`
  - ii. Comparison Operators(`==, !=, <, <=, >, >=`)
  - iii. `+, ++` (both postfix and prefix)
  - iv. `-, --` (both postfix and prefix)

❑ **Adding/Subtracting integer to a pointer**

- When an integer `n` is added/subtracted to a pointer, the amount added/subtracted is equal to `n` times the size(in bytes) of the datatype of the pointer

Pointer Arithmetic(cont…)

- e.g.
- `int i=56;`
- `int*ptr=&i;`
- `ptr++;` //this will add `1*4`(size of `int` in bytes) to `ptr`
- `ptr=ptr+2;` //this will add `2*4` to `ptr`
- `ptr=ptr-3;` //this will subtract `3*4` to `ptr`
- `(*ptr)++` //this will first dereference the pointer then do `++`

```
//Program to understand Pointer Arithmetic
#include<stdio.h>
int main(){
  int i=56; int* ptr=&i;
  printf("Value of ptr is %p\n",ptr);
  printf("Value of ptr++ is %p\n",ptr++);
  printf("Value of ptr-- is %p\n",ptr--);
  printf("Value of ptr-3 is %p\n",ptr-3);
  printf("Value of (*ptr)++ is %d\n",(*ptr)++);
  printf("Value of (*ptr)-2 is %d\n",(*ptr)-2);
  printf("Value of *ptr+3 is %d\n",*ptr+3);   /*this will
first dereference pointer as precedence of * is more than +*/
  printf("Value of *ptr++ is %d\n",*ptr++);
/*precedence of * is more than ++, this will do *ptr then do
++ on it*/
  return 0;}
```

Pointer Arithmetic(cont…)

- The Output of the above program is:

```
Value of ptr is 0x7fffb3bb0eac
Value of ptr++ is 0x7fffb3bb0eac
Value of ptr-- is 0x7fffb3bb0eb0
Value of ptr-3 is 0x7fffb3bb0ea0
Value of (*ptr)++ is 56
Value of (*ptr)-2 is 55
Value of *ptr+3 is 60
Value of *ptr++ is 57
```

Pointers and Const Keyword

- Four cases of Pointers with `const` Keyword

Case e.g. `int i=56;`	Pointer Modifiable	Data Modifiable
1. Non-constant Pointer & Non-constant Data • `e.g. int* ptr=&i;`	✓	✓
2. Non-constant Pointer & Constant Data • `e.g. int const* ptr=&i;`	✓	✗
3. Constant Pointer & Non-constant Data • `e.g. int* const ptr=&i;`	✗	✓
4. Constant Pointer & Constant Data • `e.g. const int* const ptr=&i;` OR • `int const* const ptr=&i;`	✗	✗

Pointers and Const Keyword(Case-I)

```
#include<stdio.h>
int main(){
  int i=56;
  int* ptr=&i;
  //modifying data
  printf("Value of ++(*ptr) is %d\n",++(*ptr));
  //modifying pointer
  printf("Value of ++ptr is %p\n",++ptr);
  return 0;}
```

The output of the above Program is:

```
Value of ++(*ptr) is 57
Value of ++ptr is 0x7ffda319f370
```

Pointers and Const Keyword(Case-II)

```
#include<stdio.h>
int main(){
  int i=56;
//const int* ptr=&i;/*ptr is a pointer to a constant integer*/
  int const* ptr=&i; //second way
  //modifying data
  printf("Value of ++(*ptr) is %d\n",++(*ptr)); /* this gives error*/
  printf("Value of ++ptr is %p\n",++ptr);
  return 0;}
```

The error in the above Program is:

```
error: increment of read-only location '*ptr'
   printf("Value of ++(*ptr) is %d\n",++(*ptr));
                                      ^
```

Pointers and Const Keyword(Case-III)

```
#include<stdio.h>
int main(){
   int i=56;
   int* const ptr=&i; /*ptr is a constant pointer to an integer*/
   printf("Value of ++(*ptr) is %d\n",++(*ptr));
   //modifying pointer
   printf("Value of ++ptr is %p\n",++ptr); /*this gives error*/
   return 0;}
```

The error in the above Program is:

```
error: increment of read-only variable 'ptr'
    printf("Value of ++ptr is %p\n",++ptr);
                                    ^
```

Pointers and Const Keyword(Case-IV)

```
#include<stdio.h>
int main(){
   int i=56;
//int const* const ptr=&i; /*ptr is a constant pointer to a constant integer*/
   const int* const ptr=&i; /*second way*/
   //modifying data
   printf("Value of ++(*ptr) is %d\n",++(*ptr));
   //modifying pointer
   printf("Value of ++ptr is %p\n",++ptr);
   //both of above two statements give error
   return 0;}
```

Pointers and Const Keyword(Case-IV)

The error in the above Program is:

```
error: increment of read-only location ‘*ptr’
   printf("Value of ++(*ptr) is %d\n",++(*ptr));
                                       ^
error: increment of read-only variable ‘ptr’
   printf("Value of ++ptr is %p\n",++ptr);
                                   ^
```

1D Integer Arrays & Pointers

- A 1D array can be declared like
- `datatype array-name[size];`
- e.g.
- `int arr[5]={1,2,3,4,5};` /*initialized in single statement*/
- An array is allocated memory in consistent memory locations
- The following is an example showing `arr` in memory

Starting Byte Address	100	104	108	112	116
Elements	1	2	3	4	5

1D Integer Arrays & Pointers(cont…)

- Access the elements of an array
-
```
for(int i=0;i<5;i++)
  printf("%d\t",arr[i]); //OR
  printf("%d\t",(arr+i));
```
- Note that array names are just like constant pointers, i.e. we can't change their value e.g. `arr=arr+i;` is wrong
- We can use pointer in place of array names by storing the array address in a pointer, like
- `int*ptr=arr;`
- and then use `ptr` in place of `arr`

1D Integer Arrays & pointers(cont…)

```
#include<stdio.h>
int main(){
  int arr[5]={1,2,3,4,5};
  printf("Using Array name and subscript\n");
  for(int i=0;i<5;i++)
    printf("%d\t",arr[i]);
  printf("\nUsing array name and pointer arithmetic\n");
  for(int i=0;i<5;i++)
    printf("%d\t",*(arr+i));
```

1D Integer Arrays & pointers(cont…)

```
  int*ptr=arr;
  printf("\nUsing Pointer and subscript\n");
  for(int i=0;i<5;i++)
    printf("%d\t",ptr[i]);
  printf("\nUsing Pointer and arithmetic\n");
  for(int i=0;i<5;i++)
    printf("%d\t",*(ptr+i));
  printf("\n");
  return 0;
}
```

1D Integer Arrays & Pointers(cont…)

The output of the above Program is:

```
Using Array name and subscript
1  2    3    4    5
Using array name and pointer arithmetic
1  2    3    4    5
Using Pointer and subscript
1  2    3    4    5
Using Pointer and arithmetic
1  2    3    4    5
```

2D Integer Arrays & Pointers

- A 2D array can be declared like
- `datatype array-name[rows][cols];`
- e.g.
- `int matrix[4][3]={1,2,3,…,12};` //`4` rows and `3` columns
- A 2D array is also allocated memory in consistent memory locations, at first the elements of 1st row are placed in memory and then the elements of the 2nd row and so on

Element Index	00	01	02	10	11	12	20	21	22	30	31	32
Row#	1			2			3			4		
Starting Byte Address	100	104	108	112	116	120	124	128	132	136	140	144
Elements	1	2	3	4	5	6	7	8	9	10	11	12

2D Array in Matrix form

Row \ Col	0	1	2
0	$^{00}1_{100}$	$^{01}2_{104}$	$^{02}3_{108}$
1	$^{10}4_{112}$	$^{11}5_{116}$	$^{12}6_{120}$
2	$^{20}7_{124}$	$^{21}8_{128}$	$^{22}9_{132}$
3	$^{30}10_{136}$	$^{31}11_{140}$	$^{32}12_{144}$

*The superscript represents element index(row,col) and subscript represents address of starting byte

2D Integer Arrays & Pointers(cont…)

- To access the elements of a 2D array, we use a nested loop
-
```
for(int i=0;i<4;i++)
  for(int j=0;j<3;j++)
    printf("%d\t",arr[i][j]);     //subscript notation
```

❑Use of pointers with array names

- If we write
- `arr` =>address of 2D array i.e. `100` here
- `*arr` => This will dereference once and will again give the address `100`, but `arr` gives the address of 2D array which is the same as the address of the 1st 1D array so `*arr` also gives `100`

2D Integer Arrays & Pointers(cont…)

- `**arr` =>this will dereference `arr` twice and will give `1` i.e. the element at `index=1` of 1st 1D array
- `arr+1` =>this adds `12` to `arr` i.e. the size of 1st 1D array
- `*(arr+1)` =>this will give `112` i.e. the address of the 2nd 1D array
- `**(arr+1)` => this will give `4` i.e. the element at `index=1` of 2nd 1D array

❑**How to access a particular element?**

- To access, for example, the 3rd element in 2nd 1D array, we will write like

2D Integer Arrays & Pointers(cont…)

- `*(arr+1)` =>this will take us to the 2nd 1D array
- `*(*(arr+1)+3)` => this will take us to the 3rd element of 2nd 1D array, so we will use this statement to access the 3rd element in 2nd 1D array

❑**`sizeof()` operator with array name**

- `sizeof(arr)` =>will give the size of 2D array, i.e. `48`
- `sizeof(*arr)` =>will give the size of 1st 1D array, i.e. `12`
- `sizeof(**arr)` =>will give the size of 1st element of 1st array, i.e. `4`

2D Integer Arrays & Pointers(cont…)

❑2-Ways of referring to a 2D array using pointers

1. `int (*ptr)[3] = arr;`

- This says that `ptr` is a pointer to a 1D array of integers of size `3`, no. of rows(pointers) need not to be mentioned here

2. `int *ptr[4];`

- This says that `ptr` is an array of integer pointers of size 4 and each pointer is pointing to an array of integers
- We can initialize the pointers like
- `ptr[0]=matrix[0];`
- `ptr[1]=matrix[1];` and so on

2D Integer Arrays & pointers(cont…)

```
#include<stdio.h>
int main(){
  int matrix[4][3]={1,2,3,4,5,6,7,8,9,10,11,12};
  printf("Using array name and subscript\n");
  for(int i=0;i<4;i++){
    for(int j=0;j<3;j++)
      printf("%d\t",matrix[i][j]);
    printf("\n");}
  printf("Using array name and pointer arithmetic\n");
  for(int i=0;i<4;i++){
    for(int j=0;j<3;j++)
      printf("%d\t",*(*(matrix+i)+j));
    printf("\n");}
```

2D Integer Arrays & pointers(cont…)

```
int (*ptr)[3]=matrix;
printf("Using pointer and subscript\n");
for(int i=0;i<4;i++){
  for(int j=0;j<3;j++)
    printf("%d\t",ptr[i][j]);
  printf("\n");}
printf("Using pointer and pointer arithmetic\n");
for(int i=0;i<4;i++){
  for(int j=0;j<3;j++)
    printf("%d\t",*(*(ptr+i)+j));
  printf("\n");}
return 0;}
```

2D Integer Arrays & Pointers(cont…)

The output of the above Program is:

```
Using array name and subscript
1  2    3
4  5    6
7  8    9
10 11   12
Using array name and pointer arithmetic
1  2    3
4  5    6
7  8    9
10 11   12                                    ^
```

//continued on next slide

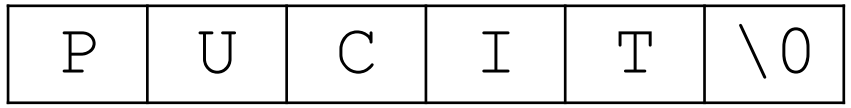
2D Integer Arrays & Pointers(cont…)

//previous output continued

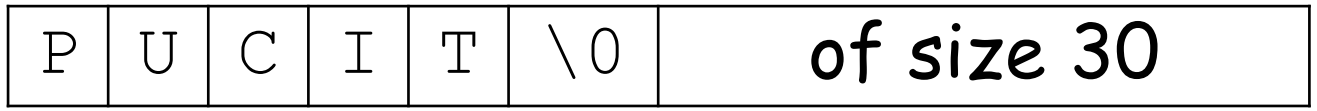
```
Using pointer and subscript
1  2    3
4  5    6
7  8    9
10 11   12
Using pointer and pointer arithmetic
1  2    3
4  5    6
7  8    9
10 11   12                              ^
```

1D Character Arrays & Pointers

- A character array is termed as a sequence of characters and a null character
- A character array can be declared like
- `char name[]="PUCIT";`



- `char name[30]="PUCIT";`



- You can input a string using functions `scanf()`, `gets()`, or take input character by character
- e.g.
- `char name[100];`
- `scanf("%[^\n]s",name);`

1D Character Arrays & Pointers(cont…)

- `fgets(name,100,stdin);`
- `while((name[i++]=getchar())!='\n');` and then `name[--i]='\0';` /*to store a null character at the end of the string*/
- One thing that you should not do is
- `char* name;`
- `scanf("%s",name);` //this will give **segmentation error**
- If you want to do this, you must allocate memory using `malloc()`

1D Character Arrays & Pointers(cont…)

```
#include<stdio.h>
int main(){
  char name1[]="Arif Butt";
  char name2[50];
//different ways to input a string
//scanf("%s",name2);
//scanf("%[^\n]s",name2);
//fgets(name,sizeof(name2),stdin); /*don't forget to replace \n with \0*/
  int i=0;
  while((name2[i++]=getchar())!='\n');
  name2[--i]='\0';
```

1D Character Arrays & Pointers(cont…)

```
//different ways to output a string
//printf("%s\n",name2);
//puts(name2);
  i=0;
  while(name2[i]!='\0')
//  printf("%c",name2[i++]);
//  putchar(name2[i++]);
    putc(name2[i++],stdout);
  putc('\n',stdout);
  return 0;
}
```

1D Character Arrays & Pointers(cont…)

The output of the above Program is:

```
Learning Linux is fun with Arif Butt
Learning Linux is fun with Arif Butt
```

- First line is the input given and the second line is the output line, which means that output has been shown same as the input

2D Character Arrays and Pointers

- A 2D character array can be declared like
- `char names[3][30]={“Arif Butt”,“Rauf”,“Jamil”};`
- It says that names has three character pointers each pointing to a separate array of characters of size 30

	0 1 2 3……30
0	Arif Butt\0 . . .
1	Rauf\0 . . .
2	Jamil\0 . . .

2D Character Arrays and Pointers(cont…)

❑Array of Pointers

- We can also use array of pointers for 2D character arrays
- e.g.
- `char* name[3]={"Arif", "Rauf", "Jamil"};`
- It declares three character pointers, each holding a different string
- Never do like this
- `char* names[3];`
- `scanf("%s", names[0]);` /*this will cause **segmentation fault*/**

2D Character Arrays and Pointers(cont…)

```
#include<stdio.h>
#include<string.h>
int main(){
//char* names[3];
  char names[3][30]; char* q;
  printf("Enter three names:\n");
  for(int i=0;i<3;i++){
    fgets(names[i],30,stdin);
    if((q=strchr(names[i],'\n'))!='\0')
      *q='\0';}
  printf("The three names entered are:\n");
  for(int i=0;i<3;i++)
    printf("%s\n",names[i]);
  return 0;}
```

2D Character Arrays and Pointers(cont…)

The output of the above Program is:

```
Enter three names:
Arif Butt
Rauf
Kakamanna
The three names entered are:
Arif Butt
Rauf
Kakamanna
```

- **Note:** `strchr()` used in the above program is a function that returns a pointer to the string where the specified character exists

SUMMARY