AROR UNIVERSITY OF ART, ARCHITECTURE, DESIGN & HERITAGE SUKKUR

Operating System

Lab-05

Compiled by: Aurangzeb Magsi

Course Title:

Lab-5

OperatingSystem

Dynamic Memory Allocation and Pointers in Linux using C

1. What are 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 using it to store any variable address. The general form of a pointer variable declaration is –

type *var-name;

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable

Example:

#include <stdio.h>

int main () {

int var = 20;

int *ip; /* pointer variable declaration */

ip = &var; /*The **address-of operator (`&`)** is used to get the **memory address of `var`**. This address is then stored in the pointer variable `ip`. Now, `ip` holds the memory location where `var` is stored.*/

printf("Address of var variable: %p\n", &var); */Prints the **memory address of `var`** using the format specifier `%p` */

printf("Address stored in ip variable: %p\n", ip); */ Prints the **value stored in `ip`**, which is the **memory address of `var`**. Since `ip = &var`, both this and the previous line should print the same address.*/

printf("Value of *ip variable: %d\n", *ip); */ `*ip` is **dereferencing the pointer**, meaning it retrieves the value stored at the memory location held by `ip`. Since `ip` stores the address of `var`, `*ip` gives the **value of `var`**, which is `20`*/

return 0; */ The `main()` function returns `0`, indicating **successful execution**.*/

}

2. NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer.

Example:
#include <stdio.h>
int main () {
int *ptr = NULL; */Declares a **pointer variable** `ptr` that stores the address of an integer. It is initialized with `NULL`, which means it **does not currently point to any valid memory address**. `NULL` is typically defined as `0` or `(void*)0` in C, indicating an **invalid or uninitialized pointer**.*/

printf("The value of ptr is : %p\n", ptr); */Prints the **value stored in** `ptr`, which is `NULL` (often displayed as `0x0` or `(nil)`). `%p` is the **format specifier for printing pointer addresses**.*/

return 0;
}

LAB TASK:

Task 1: Pointer to Multiple Variables

Write a program that:

1. Declares **two integer variables** (`a` and `b`) and initializes them.
2. Declares a **pointer** and assigns it the address of `a`.
3. Prints the **address and value** of `a` using the pointer.
4. Changes the pointer to store the address of `b`.
5. Prints the **address and value** of `b` using the pointer.

Task 2: Using a NULL Pointer and Checking Validity

Write a program that:

1. Declares an **integer pointer** and initializes it to `NULL`.
2. Prints the value of the pointer.
3. Uses an `if` condition to check if the pointer is `NULL` before dereferencing.
4. If the pointer is `NULL`, print `"Pointer is NULL, cannot dereference."`
5. Otherwise, assign it the address of an integer variable and print its value.

3. Pass by Reference:

A reference parameter "refers" to the original data in the calling function. Thus any changes made to the parameter are ALSO MADE TO THE ORIGINAL variable. Arrays are always pass by reference in C. Any change made to the parameter containing the array will change the value of the original array.

Example: Pass by reference
#include <stdio.h>

void doit(int * x){ */This function accepts a **pointer to an integer** (`int *x`), meaning it operates on the **original variable's memory. I**nside the function.*/

`*x = 5;` */ Inside the function, `*x = 5;` **dereferences** the pointer and modifies the actual value stored at that memory location.*/

}

int main(){

int z = 27;
doit(&z); */ passing the **address of `z`** (`&z`) to the function. Inside `doit()`, the value at that memory address (`z`) is modified to `5` */
printf("z is now %d\n", z); */ prints the updated value*/
return 0;
}

Example: Pass by reference (Array)
#include <stdio.h>
void build_array(int array_variable[], int length_of_array) { */ Defines a function which takes **two parameters**: An **integer array** (passed by reference) and The **size of the array**. */ for (int i=0; i<length_of_array; i++) */ A `for` loop iterates from `0` to `length_of_array - 1` (i.e., `0` to `14*/`
{
array_variable[i] = i; */), It assigns **each index** of `array_variable` the value of `i`, This effectively **fills the array** with values `{0, 1, 2, 3, ..., 14}*/`
}
}
int main(){

int values[15]; */Declares an integer array **`values`** of size `15`. The array is **uninitialized**, so it may contain **garbage values** before modification.`*/`

printf("the value at location 7 starts as %d\n", values[7]); */ Prints the **initial value** of `values[7]` (before modification).*/
build_array(values, 15); */ Calls the function `build_array()`, passing: `values` (the array) and `15` (the array size). Since **arrays are passed by reference**, `values[]` in `main()` is **directly modified** inside `build_array()`*/
printf("the value at location 7 is now %d\n", values[7]); */Prints the **new value** of `values[7]` after `build_array()` modifies the array. Since `build_array()` sets `values[i] = i`, the expected output
for `values[7]` is `7`.*/ return 0;
}

4. Dynamic Memory Allocation

Dynamic memory allocation refers to managing system memory at runtime. Dynamic memory management in C programming language is performed via a group four functions named malloc(), calloc(), realloc(), and free(). These four dynamic memory allocation functions of the C programming language are defined in the C standard library header file . Dynamic memory allocation uses the heap space of the system memory.

Malloc():

“malloc” or “memory allocation” is used to allocate a block of memory on the heap. Specifically, malloc() allocates the user a specified number of bytes but does not initialize. Once allocated, the program accesses this block of memory via a pointer that malloc() returns

```
int *ptr = (int *) malloc(SIZE_USER_NEEDS * sizeof(int));
```

Example: Memory Allocation (malloc)

```
#include <stdio.h>
#include <stdlib.h>
int main(){
int* ptr;
int n, i;
printf("Enter number of elements:");
scanf("%d",&n);
printf("Entered number of elements: %d\n", n);
ptr = (int*)malloc(n * sizeof(int));
if (ptr == NULL) { printf("Memory not allocated.\n");
}
else {
printf("Memory successfully allocated using malloc.\n");
for (i = 0; i < n; ++i) {
 ptr[i] = i + 1;
}
printf("The elements of the array are: ");
for (i = 0; i < n; ++i) {
printf("%d ", ptr[i]);
}
}
return 0;
 }
```

5. Lab- Exercises

calloc():

"calloc" or "contiguous allocation" method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value '0'.

```
ptr = (float*) calloc(25, sizeof(float));
```

• Implement Example 3.5 using "calloc"

realloc():

"realloc" or "re-allocation" method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically reallocate memory.

```
ptr = realloc(ptr, newSize);
```

Example: Memory Reallocation (realloc)

```
#include <stdio.h>
#include <stdlib.h>
int main() {
int i, * ptr, sum = 0;
ptr = malloc(100);
if (ptr == NULL) {
 printf("Error! memory not allocated.");
exit(0);
}

ptr = realloc(ptr,500);
if(ptr != NULL) printf("Memory created successfully\n");
return 0;
}
```

free()

"free" method in C is used to dynamically de-allocate the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

6. Lab- Exercises

Implement free() : Syntax: free(ptr);