Pointer in 'C'

Dwaipayan Biswas,
CSE [2019], GCETTB

Memory in 'C'

- In 'C' every variables occupied in a 1 Dimension like memory interface,
- Every variables takes memory of different length according to the types,
- Every memory has their own addresses,
- It is possible to access this variables using the address rather than variable name,

Pointer

- Pointer is a variable, which store the address of another variable,
- A variables can be different size, but pointer size is same as architecture defined [NB: Same as Integer size].
- Pointer hold the base address of a variable,
- Pointer data types is used in address arithmetic,
- & -> address operator, * -> dereferencing or indirection operator
- Example: *pointer_basic.c*

Advantages of Pointer

- It is faster to access through pointers compare than variable name,
- In dynamically created DS naming is not possible, where pointer become a good solution,
- It allows passing of arrays, strings & structure to functions more efficiently,
- It makes possible to return more than one value from a function,

Pointer & Function

- Two ways to pass parameter in function:
  - a. call by value, [func(a, b)]
  - b. call by reference, [func(&a, &b)]
- Example with swap functions

Pointer, Array & Address Arithmetic

- Array name can be act as a pointer,
- A Pointer is a variable, but array name is Numonics,
- Suppose, int a[5], where int = 2 Byte,
- *a = a[o],
- *(a + i) = a + (i * sizeof(type of 'a')),

A[0]		A[1]		A[2]		A[3]		A[4]	
100	101	102	103	104	105	106	107	108	109

- (a + i) = &a[i]
- int *p; p = &a[o]; then,
- (p+i) = &a[i]; *(p+i) = a[i] = p[i];
- (p+i) = p+(i*sizeof(typeof 'p')) = (a+i) = &a[i]
- arr[index] & index[arr] both are same

Pointer operations

- Valid operations:
  - Assignment of pointers of same type,
  - Adding or subtracting a pointer and an int,
  - Subtracting or comparing two pointers to members of same array,
  - Assigning or comparing to zero.
- Example: pointer_array.c

Not Allowed	Allowed
arr++	ptr++
arr = arr+1	ptr = ptr+1
arr = ptr	ptr = arr

Character arrays & Pointers

- Defining array of characters
- char a[] = "GCETTB"; [modifiable]
- char *p = "GCETTB"; [not modifiable, because it's a string constant]
- Example:

```
1. strcpy(char *destination, char *source)
2. {
3.          while(*s++ = *t++);
4. }
```

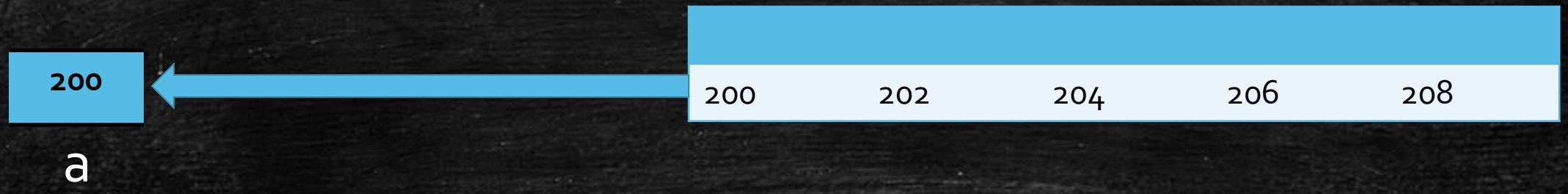
- ['strcpy' is vulnerable function, use strncpy(char *, char *, n), where n is size of source string].

Array of Pointers

- As pointers are variables it can be declared as array of Pointers,
- Array of Pointer is similar to multidimensional array,
- Array of Pointer is more space efficient than the multidimensional array,
- Example: array_pointers.c

Array in function arg.

- func(int a[5]) || func(int a[]) || func(int *a)



Dynamic memory allocation

- There is many ways memory can be allocated,

1. Static : Allocate on compile time. Global & static variables stored BSS.
2. Local : Allocate on function call. Stored in Stack. Local variables.
3. Dynamic : Allocate on runtime, stored in Heap.

malloc()

- void * malloc(size);
- “malloc” or “memory allocation” method in C is used to dynamically allocate a single large block of memory with the specified size.
- It initializes each block with default garbage value.
- *ptr = (data-type *)malloc(byte-size);
- Example : malloc.c

calloc()

- void * calloc(n, block-size);
- “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 = (data-type*) calloc(n, element-size);
- Example: calloc.c

realloc()

- void * realloc(ptr, newSize)
- If the memory previously allocated with the help of malloc or calloc is insufficient,
- Realloc can be used to dynamically re-allocate memory.
- *ptr = realloc(ptr, n * ByteSize);
- It's a very slow process,
- Example : realloc.c

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.
- Example : free.c

Segmentation fault

- Core Dump/Segmentation fault is a specific kind of error caused by accessing memory that “does not belong to you.”
- When a piece of code tries to do read and write operation in a read only location in memory or freed block of memory.
- It is an error indicating memory corruption.

Segmentation fault scenarios :

- Modify a single literal,
- Accessing an address that is freed,
- Accessing out of array index,
- Improper use of scanf();
- Dereferencing uninitialized pointer,

Dynamic mem. alloc. adv & dis adv

- Advantages:
  - When we do not know how much amount of memory would be needed for the program beforehand.
  - When you want to use your memory space more efficiently.
- Disadvantages:
  - Out of space condition may be occur after a huge memory allocation.
  - Memory leak may occur for bad memory handling.
  - Security of memory may be hampered if memory handling is not efficient.
  - Segmentation fault may occurred.

Dangling Pointer

- Pointer pointing to non-existing memory location is called dangling pointer.
- 3 different ways where Pointer acts as dangling pointer:
  - **De-allocation of memory,**
  - **Function Call,**
  - **Variable goes out of scope.**

Void Pointer

- Void pointer is a specific pointer type – void * – a pointer that points to some data location in storage, which doesn't have any specific type.
- Void refers to the type. Basically the type of data that it points to is can be any.

Null Pointer

- NULL Pointer is a pointer which is pointing to nothing. In case, if we don't have address to be assigned to a pointer, then we can simply use NULL.

```
1. #include <stdio.h>
2. int main()
3. {
4.     int *ptr = NULL;
5.     printf("The value of ptr is %p", ptr);
6.     return 0;
7. }
```

Memory leak

- Memory leak occurs when programmers create a memory in heap and forget to delete it.
- Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminate.
- To avoid memory leaks, memory allocated on heap should always be freed when no longer needed.

```
1. void f() {
2.     int *ptr = (int *) malloc(sizeof(int));
3.     return;
4. }
```

Any Question ?

Thank You