

Stack And Static Arrays

Static Arrays are stored in stack section of memory:

We know arrays get stored in contiguous memory allocation (i.e., adjacent memory allocation).

The concept here is storing array's element:

If a [0] gets stored in address :1000 of stack

The next arrays index will be 1004 of stack as address goes with the equation:

$$\text{Address}(\text{arr}[i]) = \text{Base Address} + i \times \text{sizeof}(\text{type}).$$

Here type is int , has 4 bytes size in both x64 architecture and x32 bit architecture .

We take x64 bit architecture as for today its common.

$$\rightarrow \text{Address}(\text{arr}[0]) = 1000 + 0 \times 4 \text{ bytes} = 1000$$

$$\rightarrow \text{Address}(\text{arr}[1]) = 1000 + 1 \times 4 \text{ bytes} = 1004$$

$$\rightarrow \text{Address}(\text{arr}[2]) = 1000 + 2 \times 4 \text{ bytes} = 1008$$

$$\rightarrow \text{Address}(\text{arr}[3]) = 1000 + 3 \times 4 \text{ bytes} = 1012$$

$$\rightarrow \text{Address}(\text{arr}[4]) = 1000 + 4 \times 4 \text{ bytes} = 1016$$

$$\rightarrow \text{Address}(\text{arr}[5]) = 1000 + 5 \times 4 \text{ bytes} = 1020$$

| | | | |
|-------------|------|---|---|
| $a[5] = 15$ | 1020 | } | Contiguous Memory Allocation In Stack by Static Arrays. |
| $a[4] = 14$ | 1016 | | |
| $a[3] = 13$ | 1012 | | |
| $a[2] = 12$ | 1008 | | |
| $a[1] = 11$ | 1004 | | |
| $a[0] = 10$ | 1000 | | |

As we see both stack and Arrays are linearly structured, one after the other hence they fall in linear data structure. And we can represent stack operation through arrays.

- ***Contiguous means:***
→ *Memory locations are placed next to each other without gaps.*
- ***In arrays:***
→ *All elements are stored in sequential memory addresses.*

Notice:

- *No gaps*
- *Each next element is placed immediately after the previous one*
- *Address increases by `sizeof(int)`*

That is contiguous memory.

Why Is Contiguous Allocation Important?

Because it allows:

✓Constant time access

No traversal needed.

✓Cache – friendly behavior

CPU fetches nearby memory efficiently.

✓Simple pointer arithmetic

arr + i works because memory is contiguous.

That is instead of arr[i] to fetch the value ,we can fetch it through

**** (arr + i).***

| Expression | Meaning |
|------------|--------------------------|
| arr | Address of first element |
| arr + i | Address of i-th element |
| *(arr + i) | Value at i-th element |
| arr[i] | Same as *(arr + i) |

→ Contiguous memory allocation is a property of:

- Arrays (static or dynamic)***
- Not specifically stack***

→ Array can be contiguous in:

- Stack (local array)***
- Data segment (static/global array)***
- Heap (malloc)***

→ Contiguous describes layout – not memory region.

Contiguous does NOT tell us:

- *Whether it is in stack*
- *Whether it is in heap*
- *Whether it is in data segment*

It only tells us:

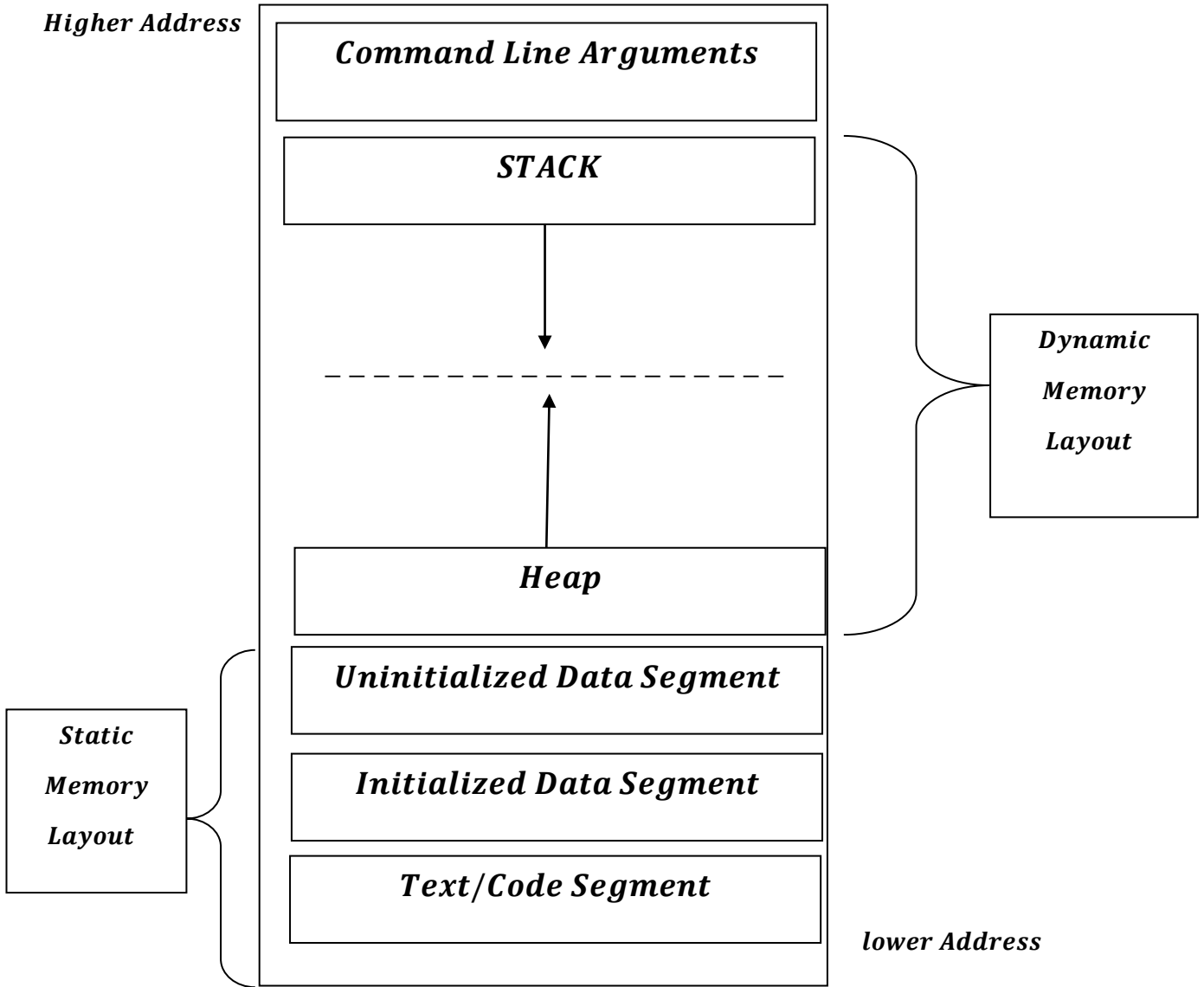
- *Elements are placed sequentially in memory.*

Contiguous memory allocation means:

All elements are stored in consecutive memory blocks with no gaps between them.

It does NOT mean:

- *Stored in stack*
- *Stored in data segment*
- *Address increments by 1*



```
void func() {  
    int arr[5];  
}
```

*Here arr[5] ,is stored in STACK , i. e. function's stack frame.
and*

```
void func()  
{  
    int *arr = malloc(5 * sizeof(int));  
}
```

*Here , functions stack frame is created , where ,
arr is the variable stored in the stack obtaining the
address of the memory allocated in the heap.*

Why Stacks created ?

- **Local automatic variables**
- **Function parameters**
- **Return address**
- **Temporary data**

All of these:

- ✓ **Created when function starts**
- ✓ **Destroyed when function ends**

```
void func()
{
    static int arr[5];
}
```

Here,

```
static int arr[5];
```

*Stored in Data Segment as the array is static not in Stack.
Allocates contiguous memory locations in data segments.*

Because stack memory:

- *Is temporary*
- *Follows LIFO*
- *Gets cleared when function exits*

But static variable:

- *Must survive after function returns*
- *Must remember previous value*

While ,

```
void func(){...}
```

the function call of func() creates a stack frame.

```
int arr[5];
```

Simple Declaration of the arr[5] also stored in Data Segements not in stack.

Stack (Automatic Storage)

- *Created when function is called*
- *Destroyed when function returns*
- *Temporary*
- *LIFO behavior*

Data Segment (Static Storage)

- *Allocated once when program starts.*
- *Exists until program terminates.*
- *NOT destroyed when function returns.*

After program terminates ,we have to remember that:

When a program runs:

- *OS creates a process*
- *Allocates virtual memory for it*
- *Sets up:*
 - *Code segment*
 - *Data segment*
 - *Heap*
 - *Stack*

When the program ends (normal exit or crash):

- The process is destroyed***
- The OS removes its page table***
- All memory is returned to the system***

So nothing remains in RAM permanently.

| <i>Memory Type</i> | <i>Lifetime</i> |
|------------------------------|------------------------------------|
| <i>Stack</i> | <i>Until function returns</i> |
| <i>Heap</i> | <i>Until freed or program ends</i> |
| <i>Static/Data</i> | <i>Until program ends</i> |
| <i>Entire process memory</i> | <i>Until OS destroys process</i> |

Similarly,

| <i>Part of Function</i> | <i>Stored Where</i> |
|--|-------------------------------|
| <i>Function code</i> | <i>Code segment of Memory</i> |
| <i>Local variables</i> | <i>Stack</i> |
| <i>Parameters</i> | <i>Stack</i> |
| <i>Return address</i> | <i>Stack</i> |
| <i>Static variables</i> | <i>Data segment of Memory</i> |
| <i>Global variables</i> <i>[not part of function but program]</i> | <i>Data segment of Memory</i> |