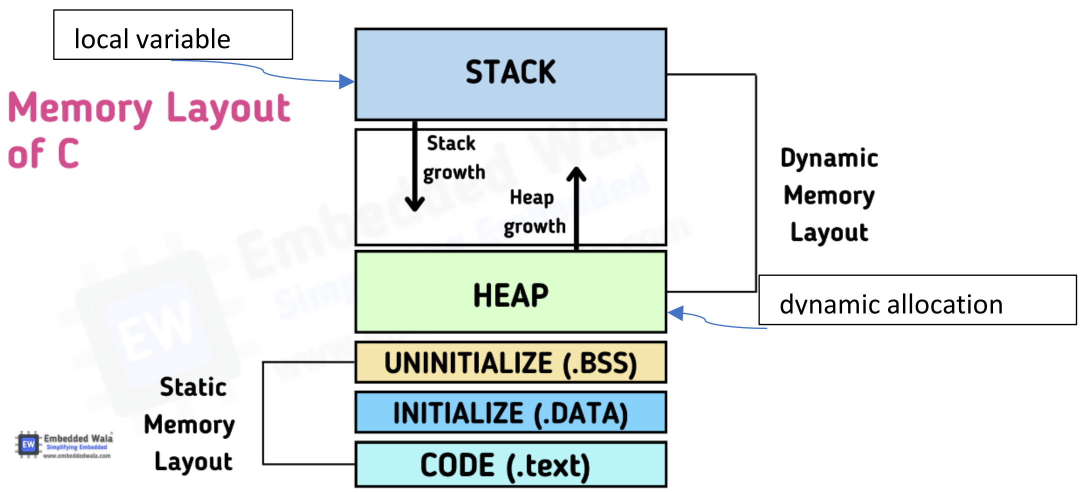
Memory Layout of C

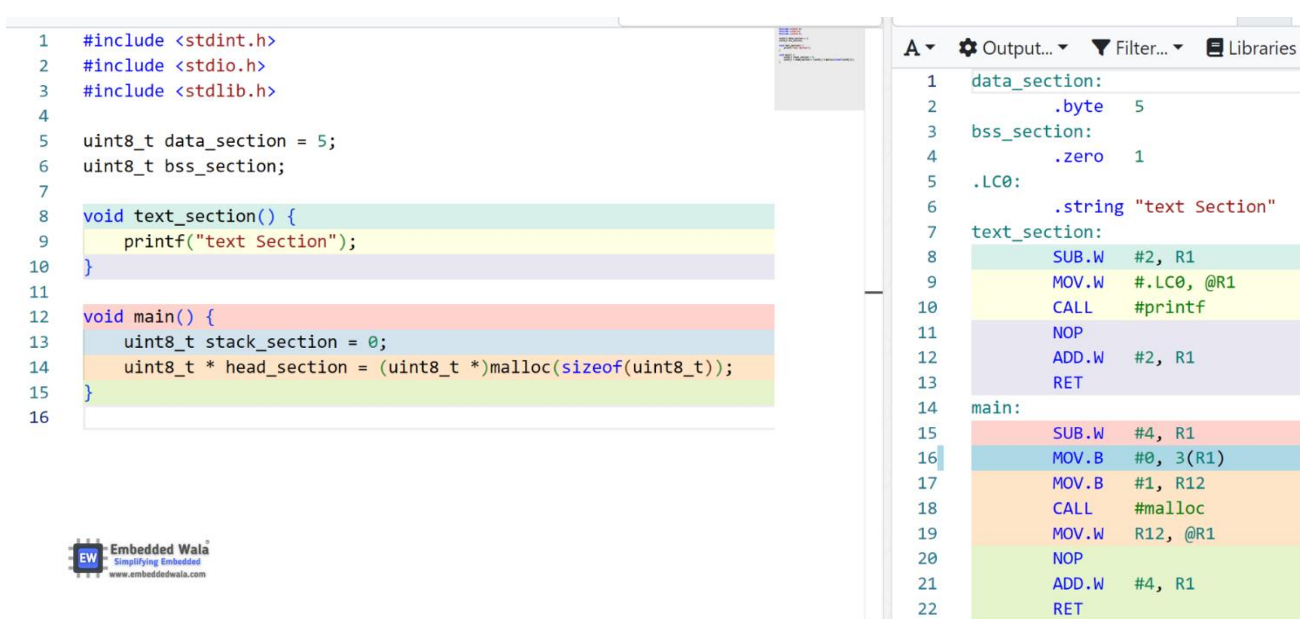
By Abdallah Ghazy

The memory layout in C programming is a fundamental concept that is crucial for understanding how memory is managed during program execution in RAM.

It defines the organization and structure of memory that the C programming language uses. In this blog post, we will delve into the memory model of C programming and its functioning.

The memory model in C programming comprises four distinct segments, each serving a particular purpose and responsible for storing different types of data. These segments are:



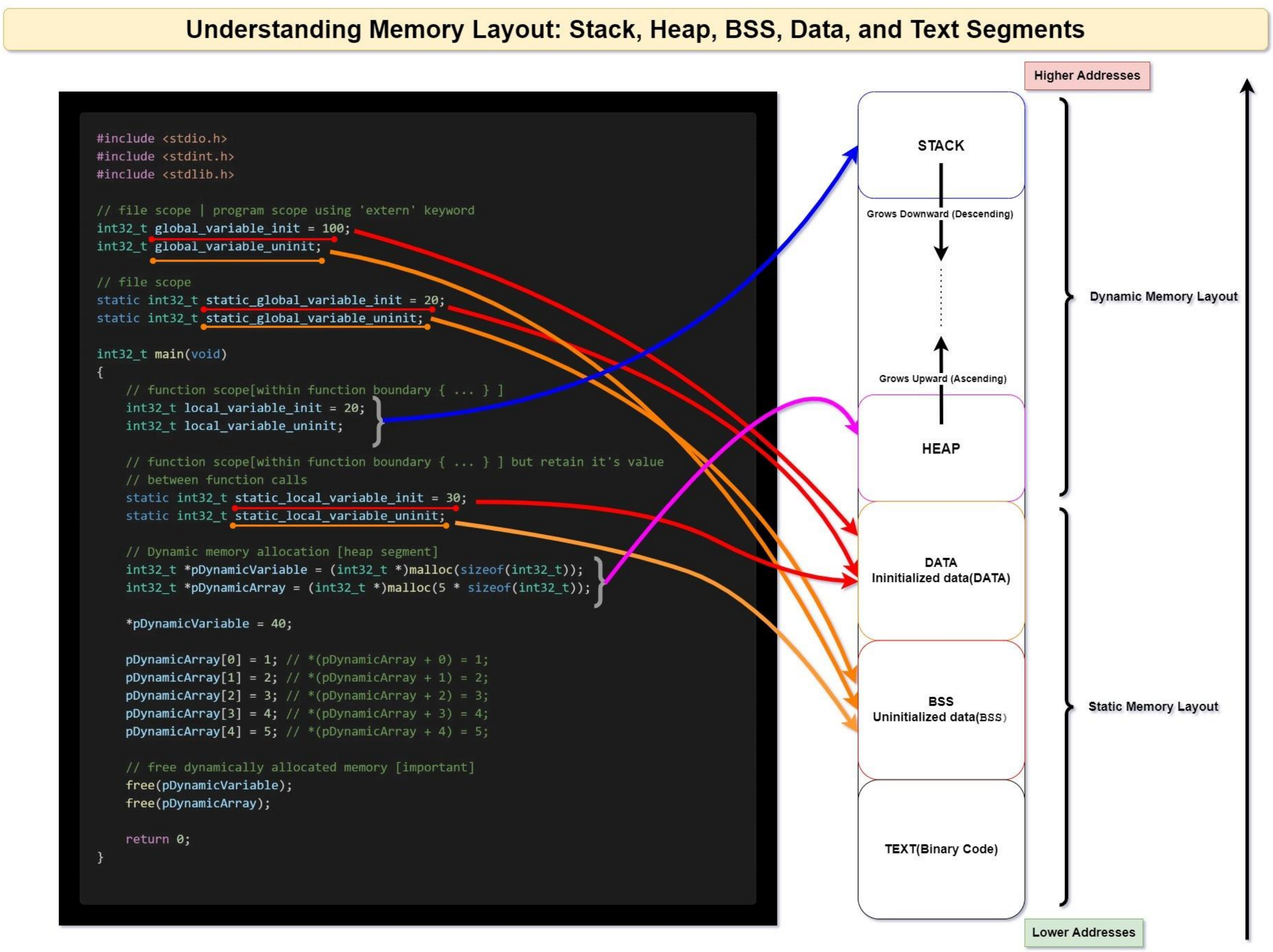


```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

uint8_t data_section = 5;
uint8_t bss_section;

void text_section() {
    printf("text Section");
}

void main() {
    uint8_t stack_section = 0;
    uint8_t * head_section = (uint8_t *)malloc(sizeof(uint8_t));
}
```

```
data_section:
        .byte   5
bss_section:
        .zero   1
.LC0:
        .string "text Section"
text_section:
        SUB.W   #2, R1
        MOV.W   #.LC0, @R1
        CALL    #printf
        NOP
        ADD.W   #2, R1
        RET
main:
        SUB.W   #4, R1
        MOV.B   #0, 3(R1)
        MOV.B   #1, R12
        CALL    #malloc
        MOV.W   R12, @R1
        NOP
        ADD.W   #4, R1
        RET
```

1. Code Segment (.text):

- ✓ The code segment, also known as the text segment, is where the compiled machine instructions of the program are stored.
- ✓ The text segment contains a binary of the compiled program
- ✓ It contains the executable code, including functions, statements, and instructions that make up the program's logic. This segment is typically read-only and shared among multiple instances of the same program.
- ✓ This is the machine language representation of the program steps to be carried out, including all functions making up the program, both user defined and system



2. Data Segment:

- There are two sub section of this segment called initialized & uninitialized data segment

➢ Initialized Data Segment (.data):

✓ This subsegment stores initialized **global** and **static** variables.

✓ Initialized variables are those that have an explicit **value assigned other than 0** during their declaration.

✓ These variables retain their values throughout the program's execution. When a variable is initialized, a memory space is allocated for it in the data segment of the program's memory.

```
#include <stdio.h>
int data1 = 10 ; //Initialized global variable stored in DS
int main(void)
{
    static int data2 = 3;  //Initialized static variable
stored in DS
    return 0;
}
```

➢ Uninitialized Data Segment (.bss):

✓ The BSS (**Block Started by Symbol**) segment stores uninitialized **global** and **static** variables.

✓ These variables are initialized to zero or null values by default.

✓ The BSS segment is zero-initialized by the operating system before the program starts executing.

✓ It is called "Block Started by Symbol" because the section starts with a symbol that defines the start of the block

```
#include <stdio.h>
int data1; // Uninitialized global variable stored in BSS
int main(void)
{
    static int data2;  // Uninitialized static variable
stored in BSS
    return 0;
}
```

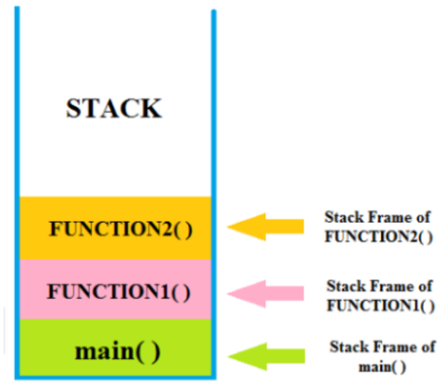
3. Stack Segment:

- ✓ The stack segment is responsible for managing function calls, local variables, and related data.
- ✓ It operates using a Last-In-First-Out (LIFO) mechanism, where the most recently called function occupies the top of the stack.
- ✓ The stack is used to store function parameters, return addresses, and local variables.
- ✓ It automatically grows and shrinks as functions are called and return.
- ✓ Each function has one stack frame.

The stack segment is area where local variables are stored. By saying local variable means that all those variables which are declared in every function including main( ) in your C program.

When we call any function, stack frame is created and when function returns, stack frame is destroyed including all local variables of that particular function.

Stack frame contain some data like return address, arguments passed to it, local variables, and any other information needed by the invoked function.



A "stack pointer (SP)" keeps track of stack by each push & pop operation onto it, by adjusted stack pointer to next or previous address.

```
#include <stdio.h>

int main(void)

{

 int data; //local variable stored in stack

return 0;

}
```

4. Heap Segment :

It is used to allocate the memory at run time.

The heap segment is the area of memory used for dynamic memory allocation. It allows programs to request memory dynamically during runtime using functions like malloc() and free(). Unlike the stack, the heap memory must be explicitly managed by the programmer. It provides flexibility in allocating and deallocating memory as needed.

```
#include <stdio.h>
int main(void)
{
    char *pStr = malloc(sizeof(char)*4); //stored in heap
    return 0;
}
```

Understanding the memory model in C programming is crucial for efficient memory management and avoiding issues like memory leaks and buffer overflows. By comprehending how data is organized and stored in memory, programmers can make informed decisions when designing and implementing their programs.

Ex

```
#include <stdio.h>

int main(void)
{
    return 0;
}

============================================================
==============
[aticleworld@CentOS]$ gcc memory-layout.c -o memory-layout
[aticleworld@CentOS]$ size memory-layout
text        data        bss        dec         hex
filename
960         248          8        1216         4c0    memory-
layout
```

- Now add a static uninitialized variable and check the size.

```
#include <stdio.h>

int main(void)
{
    static int data =10; // Stored in initialized area
    return 0;
}
============================================================
==============
[aticleworld@CentOS]$ gcc memory-layout.c -o memory-layout
[aticleworld@CentOS]$ size memory-layout
text        data        bss        dec         hex
filename
960         252          8        1216         4c0    memory-
layout
```

You can see the size of the data segment has been increased.

- Now add the global uninitialized variable and check the size.

```
#include <stdio.h>

int data; // Stored in uninitialized area

int main(void)
{
    return 0;
}
============================================================
=============
[aticleworld@CentOS]$ gcc memory-layout.c -o memory-layout
[aticleworld@CentOS]$ size memory-layout
text       data        bss        dec        hex
filename
960        248          12       1216        4c0    memory-
layout
```

You can see the size of the .bss has been increased.

- Now add the global and static uninitialized variable and check the size.

```
#include <stdio.h>

int data1; //Stored in uninitialized area

int main(void)
{
    static int data2; //Stored in uninitialized area

    return 0;
}
============================================================
==========
[aticleworld@CentOS]$ gcc memory-layout.c -o memory-layout
[aticleworld@CentOS]$ size memory-layout
text       data        bss        dec        hex
filename
960        248          16       1216        4c0    memory-
layout
```

The size of .bss increases as per the uninitialized global and static variables.

- Now add the global and static initialized variable and check the size.

```
#include <stdio.h>

int data1 = 0; //Stored in uninitialized area

int main(void)
{
    static int data2 = 0; //Stored in uninitialized area

    return 0;
}
===========================================================
==============
[aticleworld@CentOS]$ gcc memory-layout.c -o memory-layout
[aticleworld@CentOS]$ size memory-layout
text       data        bss        dec        hex
filename
960        264          8       1216        4c0    memory-
layout
```

The size of the data segment increases as per the initialized global and static variables.

" من ضيع الأصول حرم الوصول ومن ترك الدليل ضل السبيل"