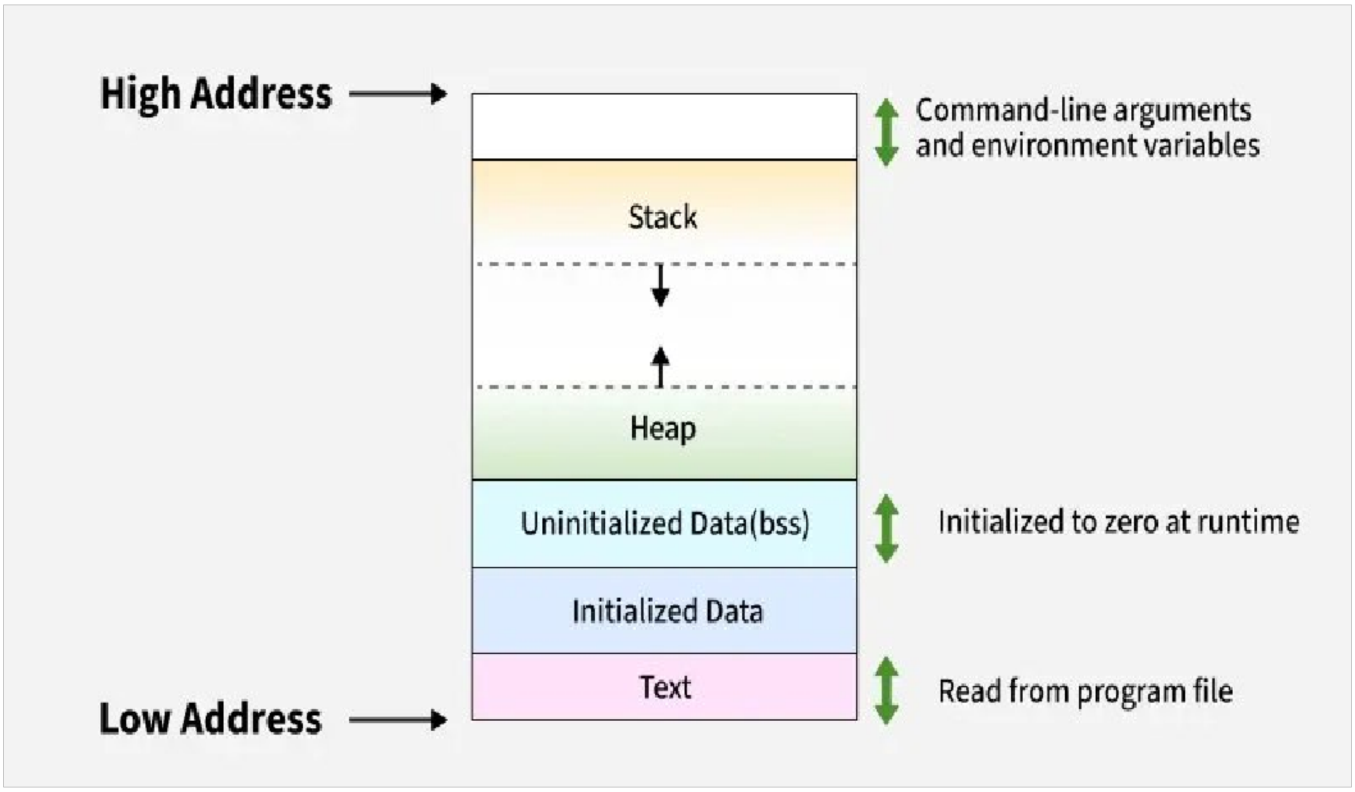
Understanding Memory Segments in C with Examples

The memory layout of a program refers to how the program's data is stored in the computer memory during its execution. Understanding this layout helps developers manage memory more efficiently and avoid issues such as segmentation faults and memory leaks.
A C program's memory is organized into specific regions (segments) as shown in the below image, each serving distinct purposes for program execution.



Different Segments in C Program's Memory

1. Text Segment

The **text segment** (also known as **code segment)** is where the executable code of the program is stored. It contains the compiled machine code of the program's functions and instructions. This segment is usually read-only and stored in the lower parts of the memory to prevent accidental modification of the code while the program is running.
The size of the text segment is determined by the number of instructions and the complexity of the program.

2. Data Segment

The **data segment** stores global and static variables that are created by the programmer. It is present just above the code segment of the program. It can be further divided into two parts:

A. Initialized Data Segment

As the name suggests, it is the part of the data segment that contains global and static variables that have been initialized by the programmer.

For example,

```
// Global variable
int a = 10;

// Static variable
static int b = 20;
```

The above variables a and b will be stored in the Initialized Data Segment.

B. Uninitialized Data Segment (BSS)

Uninitialized data segment often called the "**bss**" segment, named after an ancient assembler operator, that stood for "Block Started by Symbol" contains global and static variables that are not initialized by the programmer. These variables are automatically initialized to zero at runtime by the operating system. For example, the below shown variables will be stored in this segment:

```
// Global variable
int a;

// Static variable
static int;
```

3. Heap Segment

Heap segment is where dynamic memory allocation usually takes place. The heap area begins at the end of the BSS segment and grows towards the larger addresses from there. It is managed by functions such as malloc(), **realloc()**, and **free()** which in turn may use the brk and sbrk system calls to adjust its size.
The heap segment is shared by all shared libraries and dynamically loaded modules in a process. For example, the variable pointed by **ptr** will be stored in the heap segment:

```
#include <stdio.h>

int main() {
```

```
    // Create an integer pointer
    int *ptr = (int*) malloc(sizeof(int) * 10);

return 0;

}
```

4. Stack Segment

The **stack** is a region of memory used for **local variables** and function call management. Each time a function is called, a **stack frame** is created to store local variables, function parameters, and return addresses. This stack frame is stored in this segment.

The stack segment is generally located in the higher addresses of the memory and grows opposite to heap. They adjoin each other so when stack and heap pointer meet, free memory of the program is said to be exhausted.

Example of data stored in stack segment:

```
#include <stdio.h>


void func() {


    // Stored in the stack

    int local_var = 10;

}


int main() {

    func();

    return 0;
}
```

Practical Examples

The size(1) command in MinGW reports the sizes (in bytes) of the text, data, and bss segments of a binary file.

1. Check the following simple C program

```c
#include <stdio.h>


int main() {
    return 0;
}
```

```
gcc memory-layout.c -o memory-layout
size memory-layout
text       data        bss        dec        hex    filename
960        248          8        1216        4c0    memory-layout
```

2. Let us add one global variable in the program, now check the size of bss

```c
#include <stdio.h>


// Uninitialized variable stored in bss
int global;
int main() {
    return 0;
}
```

```
gcc memory-layout.c -o memory-layout
size memory-layout
text       data        bss        dec        hex    filename
 960        248        12        1220        4c4    memory-layout
```

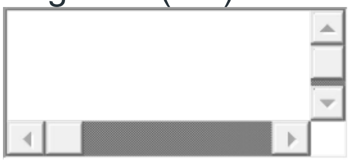
3. Let us add one static variable which is also stored in bss.

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

```
// Uninitialized variable stored in bss

int global;

int main() {


    // Uninitialized static variable stored in bss

    static int i;

    return 0;

}
```

```
gcc memory-layout.c -o memory-layout
size memory-layout
text     data     bss     dec     hex   filename
 960     248      16     1224     4c8   memory-layout
```

4. Let us initialize the static variable which will then be stored in the Data Segment (DS)



```
#include <stdio.h>


// Uninitialized variable stored in bss

int global;

int main(void) {


    // Initialized static variable stored in DS

    static int i = 100;

    return 0;

}
```

```
gcc memory-layout.c -o memory-layout
size memory-layout
text     data     bss     dec     hex   filename
960      252      12     1224     4c8   memory-layout
```

5. Let us initialize the global variable which will then be stored in the Data Segment (DS)

```c
#include <stdio.h>


// initialized global variable stored in DS

int global = 10;

int main()

    // Initialized static variable stored in DS

    static int i = 100;

    return 0;

}
```

```
gcc memory-layout.c -o memory-layout
size memory-layout
text     data     bss     dec     hex    filename
960      256      8       1224    4c8    memory-layout
```

Example to Verify the Memory Layout

```c
#include <stdio.h>

#include <stdlib.h>


// Global variable

int gvar = 66;


// Constant global variable

const int cgvar = 1010;
```

```c
// uninitialized global variable
int ugvar;

void foo() {

    // Local variable
    int lvar = 1;
    printf("Address of lvar:\t%p", (void*)&lvar);
}

int main() {

    // Heap variable
    int *hvar = (int*)malloc(sizeof(int));


    // Checking and comparing address of different
    // elements of program that should be stored in
    // different segements of the memory
    printf("Address of foo:\t\t%p\n", (void*)&foo);
    printf("Address of cgvar:\t%p\n", (void*)&cgvar);
    printf("Address of gvar:\t%p\n", (void*)&gvar);
    printf("Address of ugvar:\t%p\n", (void*)&ugvar);
    printf("Address of hvar:\t%p\n", (void*)hvar);
    foo();

    return 0;
}
```

Output

```
Address of foo:      0x60d723996189
Address of cgvar:     0x60d723997004
Address of gvar:     0x60d723999010
Address of ugvar:     0x60d723999018
Address of hvar:     0x60d73b9072a0
Address of lvar:     0x7ffd0e85e0c4
```

Comparing above addresses, we can see than it roughly matches the memory layout discussed above.

Practical Programs: Memory Segments in C

1. Code Segment (Text Segment)

This shows that function code resides in

the text segment. #include <stdio.h>

```
void demoFunction() {
   printf("Inside demoFunction\n");
}


int main() {
   printf("Address of main(): %p\n", (void*)main);
   printf("Address of demoFunction(): %p\n",
   (void*)demoFunction); return 0;
}
```

Observe that both addresses fall in the same region - the text/code segment.

2. Data Segment - Initialized Global & Static Variables

```
#include <stdio.h>
```

```
int global_var = 100;  // Initialized global
-> Data Segment static int static_var = 200;
                       // Initialized static
-> Data Segment

int main() {
   printf("Address of global_var: %p\n",
   (void*)&global_var);   printf("Address
   of          static_var:          %p\n",
   (void*)&static_var); return 0;
}
```

Both variables will be in the Initialized Data Segment.

3. BSS Segment - Uninitialized Global & Static Variables

```
#include <stdio.h>

int global_uninit;    // Uninitialized
global -> BSS static int static_uninit;
                      // Uninitialized
static -> BSS

int main() {
   printf("Address of global_uninit: %p\n",
   (void*)&global_uninit); printf("Address of
   static_uninit:                     %p\n",
   (void*)&static_uninit); return 0;
}
```

These will be in BSS segment and initialized to 0 by default.

4. Heap Segment - Using malloc

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

int main() {
   int *heap_var = (int *)malloc(sizeof(int) * 5); //
   Heap allocation

   if (heap_var != NULL) {
      printf("Address of heap_var: %p\n",
      (void*)heap_var); free(heap_var);
   }
   return 0;
}
```

Shows that dynamic memory resides in the heap.

5. Stack Segment - Local Variables

```
#include <stdio.h>

void func() {
   int local_var = 10;    // Local
```

```
    variable -> Stack printf("Address of
    local_var: %p\n",
    (void*)&local_var);
}

int main() {
    func(
    );
    retur
    n 0;
}
```

Every function call creates a stack frame.

6. All Segments in One Program

```
#include
<stdio.h>
#include
<stdlib.h>
// Global Variables
int g_init = 100;   // Initialized
global -> Data int g_uninit;  //
Uninitialized global -> BSS

void function() {   // Code ->
    Text Segment int stack_var =
    10;              // Local ->
    Stack printf("Stack
    var: %p\n",
    (void*)&stack_var);
```

```
}

int main() {
    static int s_init = 200;   // Initialized
    static -> Data static int s_uninit; //
    Uninitialized static -> BSS
    const int const_var = 999;     // Read-only data
    (may be in text segment) int *heap_var = malloc(10
    * sizeof(int));          // Heap

    printf("Function addr: %p\n",
    (void*)function); printf("g_init
    addr: %p\n", (void*)&g_init);
    printf("g_uninit addr: %p\n",
    (void*)&g_uninit); printf("s_init
    addr: %p\n", (void*)&s_init);
    printf("s_uninit addr: %p\n",
    (void*)&s_uninit); printf("const_var
    addr: %p\n", (void*)&const_var);
    printf("heap_var addr: %p\n",
    (void*)heap_var);

    function();

    free(heap_v
    ar); return
    0;
}
```

This program shows variables stored in all segments for easy comparison.