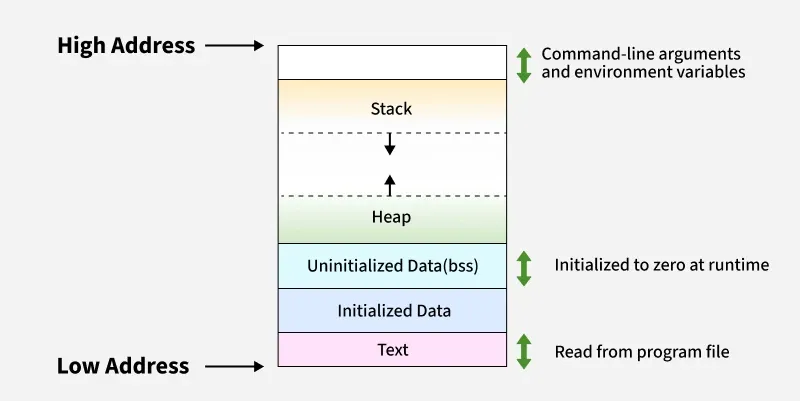
**Memory Layout of C Programs**

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 b;

**3. Heap Segment**

[**Heap**](https://www.geeksforgeeks.org/c/heap-in-c/) 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()**](https://www.geeksforgeeks.org/c/g-fact-66/), and [**free()**](https://www.geeksforgeeks.org/c/free-function-in-c-library-with-examples/) 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>

#include <stdlib.h>

​

int main() {

// Create an integer pointer

int \*ptr = (int\*) malloc(sizeof(int) \* 10);

return 0;

}

**Output**

**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;

}

**Output**

**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**

#include <stdio.h>

​

int main() {

return 0;

}

**Output**

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

#include <stdio.h>

​

// Uninitialized variable stored in bss

int global;

int main() {

return 0;

}

**Output**

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.

#include <stdio.h>

​

// Uninitialized variable stored in bss

int global;

int main() {

// Uninitialized static variable stored in bss

static int i;

return 0;

}

**Output**

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;

}

**Output**

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)

#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;

}

**Output**

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**

#include <stdio.h>

#include <stdlib.h>

​

// Global variable

int gvar = 66;

​

// Constant global variable

const int cgvar = 1010;

​

// 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: 0x400670

Address of cgvar: 0x4007a4

Address of gvar: 0x600b40

Address of ugvar: 0x600b48

Address of hvar: 0xce86010

Address of lvar: 0x7ffea42e2f3c

**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.

**Dynamic Memory Allocation in C**

Dynamic memory allocation techniques give programmer control of memory when to allocate, how much to allocate and when to de-allocate.

* Normal local variable defined in a function is stored in the [stack memory](https://www.geeksforgeeks.org/c/memory-layout-of-c-program/).
* The limitations of such allocations are, size needs to known at compile time, we cannot change the size or delete the memory.

With dynamic memory allocation,

* You allocate memory at runtime, giving you the ability to handle data of varying sizes. Dynamic resources are stored in the [**heap**](https://www.geeksforgeeks.org/c/memory-layout-of-c-program/)memory instead of the stack.
* The size of the array can be increased if more elements are to be inserted and decreased of less elements are inserted.
* The dynamically allocated memory stays there (if the programmer has not de-allocated it) even after the function call is over, so a function can return pointer to the allocated memory. In case of normal local variables/arrays, the memory/variables become invalid once the function call is over.

Dynamic memory allocation is possible in C by using the following 4 library functions provided by **<stdlib.h>** library:

**malloc()**

The **malloc()** (stands for **m**emory **alloc**ation) function is used to allocate a single block of contiguous memory on the heap at runtime. The memory allocated by malloc() is uninitialized, meaning it contains garbage values.

Assume that we want to create an array to store 5 integers. Since the size of int is 4 bytes, we need 5 \* 4 bytes = 20 bytes of memory. This can be done as shown:

#include <stdio.h>

#include <stdlib.h>

​

int main() {

int \*ptr = (int \*)malloc(20);

// Populate the array

for (int i = 0; i < 5; i++)

ptr[i] = i + 1;

// Print the array

for (int i = 0; i < 5; i++)

printf("%d ", ptr[i]);

return 0;

}

**Output**

1 2 3 4 5

In the above malloc call, we hardcoded the number of bytes we need to store 5 integers. But we know that the size of the integer in C depends on the architecture. So, it is better to use the [**sizeof operator**](https://www.geeksforgeeks.org/c/sizeof-operator-c/)to find the size of type you want to store.

#include <stdio.h>

#include <stdlib.h>

​

int main() {

int \*ptr = (int \*)malloc(sizeof(int) \* 5);

// Populate the array

for (int i = 0; i < 5; i++)

ptr[i] = i + 1;

// Print the array

for (int i = 0; i < 5; i++)

printf("%d ", ptr[i]);

return 0;

}

**Output**

1 2 3 4 5

Moreover, if there is no memory available, the malloc will fail and return NULL. So, it is recommended to check for failure by comparing the ptr to NULL.

#include <stdio.h>

#include <stdlib.h>

​

int main() {

int \*ptr = (int \*)malloc(sizeof(int) \* 5);

// Checking if failed or pass

if (ptr == NULL) {

printf("Allocation Failed");

exit(0);

}

// Populate the array

for (int i = 0; i < 5; i++)

ptr[i] = i + 1;

// Print the array

for (int i = 0; i < 5; i++)

printf("%d ", ptr[i]);

return 0;

}

**Output**

1 2 3 4 5

**Syntax**

malloc(size);

where **size**is the number of bytes to allocate.

This function returns a void pointer to the allocated memory that needs to be converted to the pointer of required type to be usable. If allocation fails, it returns NULL pointer.

**calloc()**

The**calloc()** (stands for **c**ontiguous **alloc**ation) function is similar to malloc(), but it initializes the allocated memory to zero. It is used when you need memory with default zero values.

#include <stdio.h>

#include <stdlib.h>

​

int main() {

int \*ptr = (int \*)calloc(5, sizeof(int));

// Checking if failed or pass

if (ptr == NULL) {

printf("Allocation Failed");

exit(0);

}

// No need to populate as already

// initialized to 0

// Print the array

for (int i = 0; i < 5; i++)

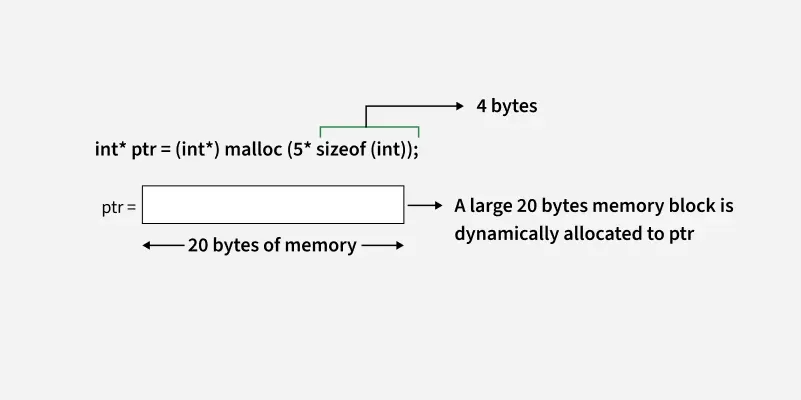
printf("%d ", ptr[i]);

return 0;

}

**Output**

0 0 0 0 0



**Syntax**

calloc(n, size);

where **n** is the number of elements and **size**is the size of each element in bytes.

This function also returns a void pointer to the allocated memory that is converted to the pointer of required type to be usable. If allocation fails, it returns NULL pointer.

**free()**

The memory allocated using functions malloc() and calloc() is not de-allocated on their own. The [**free()**](https://www.geeksforgeeks.org/c/free-function-in-c-library-with-examples/) function is used to release dynamically allocated memory back to the operating system. It is essential to free memory that is no longer needed to avoid memory leaks.

#include <stdio.h>

#include <stdlib.h>

​

int main() {

int \*ptr = (int \*)calloc(5, sizeof(int));

// Do some operations.....

for (int i = 0; i < 5; i++)

printf("%d ", ptr[i]);

// Free the memory after completing

// operations

free(ptr);

return 0;

}