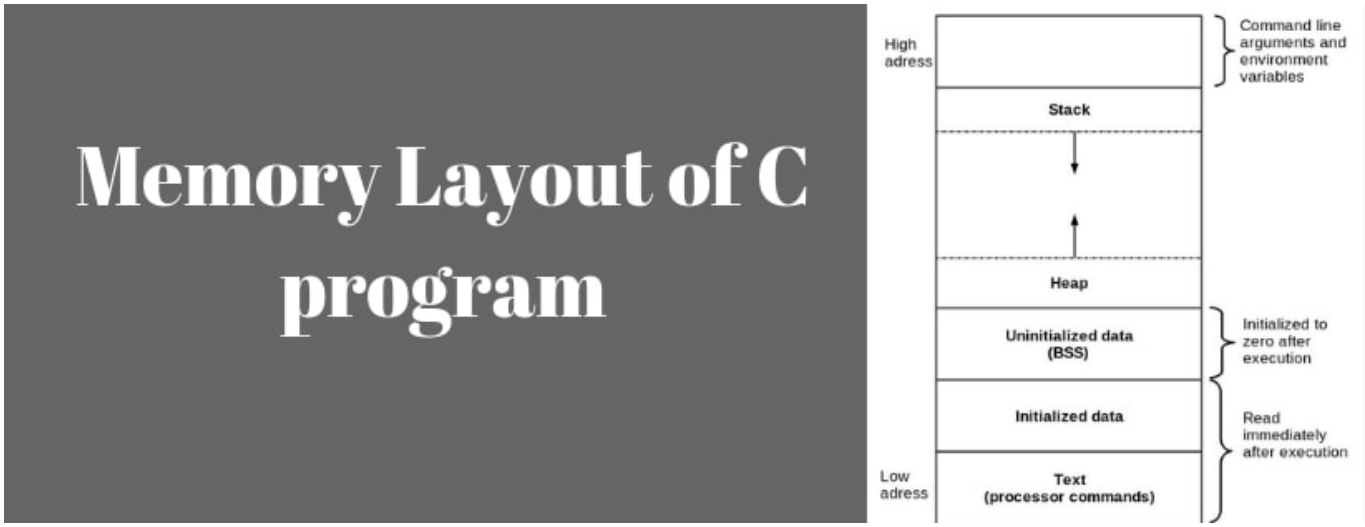
Memory Layout of C program



Basically, the memory layout of C program contains five segments these are the stack segment, heap segment, BSS (block started by symbol), DS (Data Segment) and text segment.

Each segment has own read, write and executable permission. If a program tries to access the memory in a way that is not allowed then segmentation fault occurs.

A **segmentation fault (https://aticleworld.com/10-interview-question-on-dynamic-memory-allocation/)** is a common problem that causes programs to crash. A core file (core dumped file) also associated with a segmentation fault that is used by the developer to finding the root cause of the crashing (segmentation fault).

Below find the memory Layout of C Program

Note: You must remember that this is only an example. The actual static memory layout is specific to the processor, development tools, and the underlying hardware.

1. Stack
2. Heap
3. BSS (Uninitialized data segment)
4. DS (Initialized data segment)
5. Text

```
Addresses ---> .----------------------.
               |      Environment     |
               |----------------------|
```

```
                |                       |   Functions and variable are declared
                |         STACK         |   on the stack.
 pointer ->     | - - - - - - - - - - - |
                |           |           |
                |           v           |
                :                       :
                .                       .   The stack grows down into unused space
                .         Empty         .   while the heap grows up.
                .                       .
                .                       .   (other memory maps do occur here, such
                .                       .    as dynamic libraries, and different memory
                :                       :    allocate)
                |           ^           |
                |           |           |
 point ->       | - - - - - - - - - - - |   Dynamic memory is declared on the heap
                |           HEAP        |
                |                       |
                |-----------------------|
                |           BSS         |   Uninitialized data (BSS)
                |-----------------------|
                |           Data        |   Initialized data (DS)
                |-----------------------|
                |           Text        |   Binary code
Addresses ----> '-----------------------'
```

Stack:

- It located at a higher address and grows and shrinks opposite to the heap segment.
- The stack contains local variables from functions and related book-keeping data.
- A stack frame will create in the stack when a function is called.
- Each function has one stack frame.
- Stack frames contain the function's local **variables (https://aticleworld.com/variable-in-c-language/)** arguments and return value.
- The stack contains a LIFO structure. Function variables are pushed onto the stack when called and functions variables are popped off the stack when return.
- SP(stack pointer) register tracks the top of the stack.

```c
#include <stdio.h>

int main(void)
{
    int data; //local variable stored in stack

    return 0;
}
```

Heap:

- It is used to allocate the memory at run time.
- Heap area managed by the **memory management (https://aticleworld.com/dynamic-memory-allocation-in-c/)** functions like malloc, calloc, free, etc which may internally use the brk and sbrk system calls to adjust its size.
- The Heap area is shared by all shared libraries and dynamically loaded modules in a process.
- It grows and shrinks in the opposite direction of the stack.

```
#include <stdio.h>

int main(void)
{
    char *pStr = malloc(sizeof(char)*4); //stored in heap

    return 0;
}
```

You can also see below articles,

- **Dynamic memory allocation in C (https://aticleworld.com/dynamic-memory-allocation-in-c/)**
- **Common mistakes with memory allocation (https://aticleworld.com/mistakes-with-memory-allocation/)**
- **Questions about dynamic memory allocation (https://aticleworld.com/10-interview-question-on-dynamic-memory-allocation/)**

BSS(Uninitialized data segment):

- It contains all uninitialized global and static variables.

- All variables in this segment initialized by the zero(0) and pointer with the **null pointer (https://aticleworld.com/dangling-void-null-wild-pointers/)**.
- The program loader allocates memory for the BSS section when it loads the program.

```
#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;
}
```

DS(Initialized data segment):

- It contains the explicitly initialized global and static variables.
- The size of this segment is determined by the size of the values in the program's source code and does not change at run time.
- It has read-write permission so the value of the variable of this segment can be changed at run time.
- This segment can be further classified into an initialized read-only area and an initialized read-write area.

```
#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;
}
```

Text:

- The text segment contains a binary of the compiled program.

- The text segment is a read-only segment that prevents a program from being accidentally modified.
- It is sharable so that only a single copy needs to be in memory for frequently executed programs such as text editors etc.

Note: The size command basically lists section sizes as well as total size for the input object file.

Let see few examples to understand the memory layout of the C program.

```
#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; // 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          12       1216        4c0    memory-layout
```

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

- Now add the initialized static 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.

In the data segment, I have said that the “data segment can be further classified into the two-part initialized read-only area and an initialized read-write area”. So let us see two C programs to understand this concept.

```
#include <stdio.h>

char str[]= "Amlendra Kumar";

int main(void)
{
    printf("%s\n",str);

    str[0]='k';

    printf("%s\n",str);

    return 0;
}
```

Output:

```
Amlendra Kumar
kmlendra Kumar
```

You can see the above example str is a global array, so it will go in the data segment. You can also see that I am able to change the value so it has read and write permission.

Now see the other example code,

```
#include <stdio.h>

char *str= "Amlendra Kumar";

int main(void)
{
    str[0]='k';

    printf("%s\n",str);

    return 0;
}
```

In the above example, we are not able to change the array character is because it is a literal string. A constant string does not only go in the data section but all types of const global data go in that section.

It is not necessarily that const global and constant string go in the data section. It can be also in the text section of the program (normally the .rodata segment), as it is normally not modifiable by a program.

Recommended Posts for you

- **Internal, external and none linkage in C. (https://aticleworld.com/linkage-in-c/)**
- **Create a students management system in C. (https://aticleworld.com/student-record-system-project-in-c/)**
- **Create an employee management system in C. (https://aticleworld.com/employee-record-system-project-in-c/)**
- **Top 11 Structure Padding Interview Questions in C (https://aticleworld.com/structure-padding-questions/)**
- **structure in C: you should know in depth (https://aticleworld.com/structure-in-c/)**
- **What is flexible array member in c? (https://aticleworld.com/flexible-array-member/)**
- **What is importance of struct hack in c? (https://aticleworld.com/struct-hack-in-c/)**
- **How to use the structure of function pointer in c language? (https://aticleworld.com/function-pointer-in-c/)**
- **Function pointer in structure. (https://aticleworld.com/function-pointer-in-c-struct/)**
- **Pointer Arithmetic in C. (https://aticleworld.com/pointer-arithmetic/)**
- **Union in C, A detailed Guide. (https://aticleworld.com/union-in-c/)**
- **typedef vs #define in C. (https://aticleworld.com/typedef-vs-define-in-c/)**
- **Macro in C, with example code. (https://aticleworld.com/c-macros/)**
- **enum in C, you should know. (https://aticleworld.com/seven-important-points-enum-c-language/)**
- **You should know the volatile Qualifier. (https://aticleworld.com/understanding-volatile-qualifier-in-c/)**

- 100 C interview Questions. (https://aticleworld.com/c-interview-questions/)
- Interview questions on bitwise operators in C. (https://aticleworld.com/interview-questions-on-bitwise-operators-in-c/)
- A brief description of the pointer in C (https://aticleworld.com/pointers-in-c/).
- Dangling, Void, Null and Wild Pointers (https://aticleworld.com/dangling-void-null-wild-pointers/)
- 10 questions about dynamic memory allocation. (https://aticleworld.com/10-interview-question-on-dynamic-memory-allocation/)
- File handling in C. (https://aticleworld.com/file-handling-in-c/)
- Pointer in C (https://aticleworld.com/pointers-in-c/).
- C language character set (https://aticleworld.com/character-set-and-keywords-in-c/).
- Elements of C Language (https://aticleworld.com/elements-of-c-language/).
- Data type in C language. (https://aticleworld.com/data-types-in-c-language/)
- Operators with Precedence and Associativity in C (https://aticleworld.com/operator-precedence-and-associativity-in-c/).
- C format specifiers. (https://aticleworld.com/format-specifiers-in-c/)
- C++ Interview Questions. (https://aticleworld.com/cpp-interview-questions/)

About (https://aticleworld.com/author/pritosh/)

I am an embedded c software engineer and a corporate trainer, currently, I am working as senior software engineer in a largest Software consulting company . I have working experience of different microcontrollers (stm32, LPC, PIC AVR and 8051), drivers (USB and virtual com-port), POS device (VeriFone) and payment gateway (global and first data).

12 comments

ANBU

MARCH 20, 2019 AT 5:53 PM (HTTPS://ATICLEWORLD.COM/MEMORY-LAYOUT-OF-C-PROGRAM/#COMMENT-2408)

Thank You for this awesome article

REPLY

Amlendra (https://aticleworld.com/)

MARCH 30, 2019 AT 12:28 PM (HTTPS://ATICLEWORLD.COM/MEMORY-LAYOUT-OF-C-PROGRAM/#COMMENT-2493)

Thanks, Anbu and Welcome for the suggestion.

REPLY

Afreen Khanum

APRIL 27, 2019 AT 2:28 AM (HTTPS://ATICLEWORLD.COM/MEMORY-LAYOUT-OF-C-PROGRAM/#COMMENT-2643)

You are excellent 

REPLY

Amlendra (https://aticleworld.com/)

MAY 3, 2019 AT 9:49 AM (HTTPS://ATICLEWORLD.COM/MEMORY-LAYOUT-OF-C-PROGRAM/#COMMENT-2677)

Thanks, Afreen.

REPLY

Asura

FEBRUARY 27, 2020 AT 12:40 AM (HTTPS://ATICLEWORLD.COM/MEMORY-LAYOUT-OF-C-PROGRAM/#COMMENT-4152)

Thanks for giving simple explanation. I couldn't pick up some points in class. Now that doubts are clear.

REPLY

Amlendra (https://aticleworld.com/)

FEBRUARY 27, 2020 AT 8:52 PM (HTTPS://ATICLEWORLD.COM/MEMORY-LAYOUT-OF-C-PROGRAM/#COMMENT-4153)

Thank you so much..

REPLY

Sai

MARCH 13, 2020 AT 8:57 AM (HTTPS://ATICLEWORLD.COM/MEMORY-LAYOUT-OF-C-PROGRAM/#COMMENT-4216)

Nice and informative

REPLY

Amlendra (https://aticleworld.com/)

MARCH 15, 2020 AT 10:19 PM (HTTPS://ATICLEWORLD.COM/MEMORY-LAYOUT-OF-C-PROGRAM/#COMMENT-4250)

Thanks and welcome for suggestion.

REPLY

Al Amreen

OCTOBER 7, 2020 AT 11:24 PM (HTTPS://ATICLEWORLD.COM/MEMORY-LAYOUT-OF-C-PROGRAM/#COMMENT-4882)

Thank you somuch…

REPLY

Shivaji K

APRIL 12, 2021 AT 7:37 PM (HTTPS://ATICLEWORLD.COM/MEMORY-LAYOUT-OF-C-PROGRAM/#COMMENT-5636)

Thank you so much really i have learnt so much from Article.com

REPLY

Sai sankar

JULY 11, 2021 AT 9:08 PM (HTTPS://ATICLEWORLD.COM/MEMORY-LAYOUT-OF-C-PROGRAM/#COMMENT-5850)

Bro, I have a doubt
How does text area increase.
Thanks in advance.

REPLY

Amlendra (https://aticleworld.com/)

JULY 12, 2021 AT 8:02 AM (HTTPS://ATICLEWORLD.COM/MEMORY-LAYOUT-OF-C-PROGRAM/#COMMENT-5851)

Text area depends on the binary size (code size). If the size of binary increases text area also increases. You can check the memory map (.map) file for better understanding.

REPLY

Leave a Reply

YOUR EMAIL ADDRESS WILL NOT BE PUBLISHED. REQUIRED FIELDS ARE MARKED *

COMMENT *

NAME *

EMAIL *

WEBSITE

SAVE MY NAME, EMAIL, AND WEBSITE IN THIS BROWSER FOR THE NEXT TIME I COMMENT.

POST COMMENT

(https://aticleworld.com/best-c-programming-books/)

(https://aticleworld.com/best-gift-programmers/)

Join Aticleworld

You will also get our free C interview questions eBook

Enter your email here*

Subscribe

Pages

About (https://aticleworld.com/about/)
Guest Article (https://aticleworld.com/guest-article/)
Blog Posts (https://aticleworld.com/blog-post/)
affiliate-disclosure (https://aticleworld.com/affiliate-disclosure/)
disclaimer (https://aticleworld.com/disclaimer/)