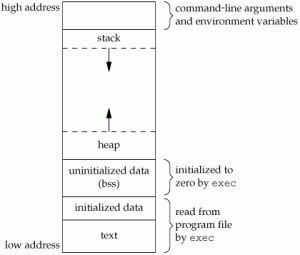
Memory Layout of C Programs

A typical memory representation of C program consists of following sections.

1. Text segment  
2. Initialized data segment  
3. Uninitialized data segment  
4. Stack  
5. Heap

[](http://d1gjlxt8vb0knt.cloudfront.net/wp-content/uploads/Memory-Layout.gif)  
A typical memory layout of a running process

**1. Text Segment:**   
A text segment, also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which **contains executable instructions**.

As a memory region, a text segment may be placed *below the heap or stack* in order to prevent heaps and stack overflows from overwriting it.

Usually, the text segment is ***sharable*** so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is **often read-only**, to prevent a program from accidentally modifying its instructions.

**2. Initialized Data Segment:**  
Initialized data segment, usually called simply the Data Segment. A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

Note that, data segment is not read-only, since the values of the variables can be altered at run time.

This segment can be further classified into initialized **read-only area** and initialized **read-write area**.

For instance the global string defined by char s[] = “hello world” in C and a C statement like int debug=1 outside the main (i.e. global) would be stored in initialized read-write area. And a global C statement like const char\* string = “hello world” makes the string literal “hello world” to be stored in initialized read-only area and the character pointer variable string in initialized read-write area.

Ex: static int i = 10 will be stored in data segment and global int i = 10 will also be stored in data segment

**3. Uninitialized Data Segment:**  
Uninitialized data segment, often called the “bss” segment, named after an ancient assembler operator that stood for “*block started by symbol*.” Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing

uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

For instance a variable declared static int i; would be contained in the BSS segment.  
For instance a global variable declared int j; would be contained in the BSS segment.

**4. Stack:**  
The stack area traditionally **adjoined the heap area** and **grows the opposite direction**; when the stack pointer met the heap pointer, free memory was exhausted. (With modern large address spaces and virtual memory techniques they may be placed almost anywhere, but they still typically grow opposite directions.)

The stack area contains the **program stack**, a LIFO structure, typically located in the higher parts of memory. On the standard PC x86 computer architecture it grows toward address zero; on some other architectures it grows the opposite direction. A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack. The set of values pushed for one function call is termed a “**stack frame**”; a stack frame consists at minimum of a return address.

Stack, where automatic variables are stored, along with information that is saved each time a function is called. **Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack.** The newly called function then allocates room on the stack for its automatic and temporary variables. This is how **recursive functions** in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn’t interfere with the variables from another instance of the function.

**5. Heap:**  
Heap is the segment where dynamic memory allocation usually takes place.

The heap area begins at the end of the BSS segment and grows to larger addresses from there. The Heap area is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size (note that the use of brk/sbrk and a single “heap area” is not required to fulfill the contract of malloc/realloc/free; they may also be implemented using mmap to reserve potentially non-contiguous regions of virtual memory into the process’ virtual address space). The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

The **size(1)** command reports the sizes (in bytes) of the text, data, and bss segments. ( for more details please refer man page of size(1) )

**size(1) - Linux man page**

The GNU **size** utility lists the section sizes---and the total size---for each of the object or archive files *objfile*in its argument list. By default, one line of output is generated for each object file or each module in an archive.

*objfile*... are the object files to be examined. If none are specified, the file "a.out" will be used.

Examples:

|  |
| --- |
| #include <stdio.h>    int main(void)  {      return 0;  } |

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 248 8 1216 4c0 memory-layout

 Let us add **one global variable** in program, now check the size of bss (highlighted in red color).

|  |
| --- |
| #include <stdio.h>    int global; /\* Uninitialized variable stored in bss\*/    int main(void)  {      return 0;  } |

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 248 **12** 1220 4c4 memory-layout

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

|  |
| --- |
| #include <stdio.h>    int global; /\* Uninitialized variable stored in bss\*/    int main(void)  {      static int i; /\* Uninitialized static variable stored in bss \*/      return 0;  } |

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 248 **16** 1224 4c8 memory-layout

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

|  |
| --- |
| #include <stdio.h>    int global; /\* Uninitialized variable stored in bss\*/    int main(void)  {      static int i = 100; /\* Initialized static variable stored in DS\*/      return 0;  } |

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 **252 12** 1224 4c8 memory-layout

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

|  |
| --- |
| #include <stdio.h>    int global = 10; /\* initialized global variable stored in DS\*/    int main(void)  {      static int i = 100; /\* Initialized static variable stored in DS\*/      return 0;  } |

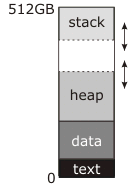
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 **256 8** 1224 4c8 memory-layout

[**What does brk( ) system call do?**](http://stackoverflow.com/questions/6988487/what-does-brk-system-call-do)



The "break"--the address manipulated by brk and sbrk--is the dotted line at the top of the *heap*. The documentation you've read describes this as the end of the "data segment" because in traditional (pre-shared-libraries, pre-mmap) Unix the data segment was continuous with the heap; before program start, the kernel would load the "text" and "data" blocks into RAM starting at address zero (actually a little above address zero, so that the NULL pointer genuinely didn't point to anything) and set the break address to the end of the data segment. The first call to malloc would then use sbrk to move the break up and create the heap *in between* the top of the data segment and the new, higher break address, as shown in the diagram, and subsequent use of malloc would use it to make the heap bigger as necessary.

Meantime, the stack starts at the top of memory and grows down. The stack doesn't need explicit system calls to make it bigger; either it starts off with as much RAM allocated to it as it can ever have (this was the traditional approach) or there is a region of reserved addresses below the stack, to which the kernel automatically allocates RAM when it notices an attempt to write there (this is the modern approach). Either way, there may or may not be a "guard" region at the bottom of the address space that can be used for stack. If this region exists (all modern systems do this) it is permanently unmapped; if *either* the stack or the heap tries to grow into it, you get a segmentation fault. Traditionally, though, the kernel made no attempt to enforce a boundary; the stack could grow into the heap, or the heap could grow into the stack, and either way they would scribble over each other's data and the program would crash. If you were very lucky it would crash immediately.

I'm not sure where the number 512GB in this diagram comes from. It implies a 64-bit virtual address space, which is inconsistent with the very simple memory map you have there. A real 64-bit address space looks more like this:

less simplified address space

This is not remotely to scale, and it shouldn't be interpreted as exactly how any given OS does stuff (after I drew it I discovered that Linux actually puts the executable much closer to address zero than I thought it did, and the shared libraries at surprisingly high addresses). The black regions of this diagram are unmapped -- any access causes an immediate segfault -- and they are *gigantic* relative to the gray areas. The light-gray regions are the program and its shared libraries (there can be dozens of shared libraries); each has an *independent* text and data segment (and "bss" segment, which also contains global data but is initialized to all-bits-zero rather than taking up space in the executable or library on disk). The heap is no longer necessarily continous with the executable's data segment -- I drew it that way, but it looks like Linux, at least, doesn't do that. The stack is no longer pegged to the top of the virtual address space, and the distance between the heap and the stack is so enormous that you don't have to worry about crossing it.

The break is still the upper limit of the heap. However, what I didn't show is that there could be dozens of independent allocations of memory off there in the black somewhere, made with mmap instead of brk. (The OS will try to keep these far away from the brk area so they don't collide.)