Memory Layout in C

When we create a C program and run the program, its executable file is stored in the RAM of the computer in an organized manner.

The memory layout for C program can be shown below:



Memory Layout in C

As we can observe in the above figure, the C program consists of the following sections in the program:

* Text segment
* Initialized data segment
* Uninitialized data segment
* Stack
* Heap

Let's understand each section one by one.

1. Text segment

The text segment is also known as the code segment. When we compile any program, it creates an executable file like a.out, .exe, etc., that gets stored in the text or code section of the RAM memory. If we store the instructions in the hard disk, then the speed for accessing the instructions from the hard disk becomes slower as hard disk works on the serial communication so taking the data from the hard disk will be slower, whereas the RAM is directly connected to the data and address bus so accessing the data from the RAM is faster.

2. Data section

The data which we use in our program will be stored in the data section. Since the variables declared inside the main() function are stored in the stack, but the variables declared outside the main() method will be stored in the data section. The variables declared in the data section could be stored in the form of initialized, uninitialized, and it could be local or global. Therefore, the data section is divided into four categories, i.e., initialized, uninitialized, global, or local.

Let's understand this scenario through an example.

#include<stdio.h>

int var1;

int var2 = 10;

void function1()

{

printf("I am function1");

}

int main()

{

function1();

return 0;

}

In the above code, var1 and var2 variables are declared outside the main() function where var1 is the uninitialized variable, whereas the var2 is an initialized variable. These variables can be accessed anywhere in the program because these variables are not a part of the main() in the stack.

The data section consists of two segments:

* Uninitialized data segment
* Initialized data segment

Uninitialized data segment

The uninitialized data segment is also known as a .bss segment that stores all the uninitialized global, local and external variables. If the global, static and external variables are not initialized, they are assigned with zero value by default.

The .bss segment stands for Block Started by symbol. The bss segment contains the object file where all the statically allocated variables are stored. Here, statically allocated objects are those objects without explicit initialization are initialized with zero value. In the above code, var1 is an uninitialized variable so it is stored in the uninitialized data section.

Let's look at some examples of uninitialized data segment.

#include<stdio.h>

char a; // uninitialized global variable..

int main()

{

static int a; // uninitialized static variable..

return 0;

}

Initialized data segment

An initialized data segment is also known as the data segment. A data segment is a virtual address space of a program that contains all the global and static variables which are explicitly initialized by the programmer.

The values of variables in a data segment are not read-one, i.e., they can be modified at run time. This data segment can be further classified into categories:

Initialized read-only area: It is an area where the values of variables cannot be modified.

Initialized read-write area: It is an area where the values of variables can also be altered.

For example: the global variables like char str[] = "javatpoint" and int a=45; will be stored in the initialized read-write area. If we create the global variable like const char\* string = "javatpoint"; the literal "javatpoint" would be stored in the initialized read area, whereas the char pointer variable would be stored in the initialized write area.

#include<stdio.h>

char string[] = "javatpoint"; // global variable stored in initialized data segment in read-write area..

int main()

{

static int i = 90; // static variable stored in initialized data segment..

return 0;

}

3. Stack

When we define a function and call that function then we use the stack frame. The variables which are declared inside the function are stored in the stack. The function arguments are also stored in the function as the arguments are also a part of the function. Such a type of memory allocation is known as static memory allocation because all the variables are defined in the function, and the size of the variables is also defined at the compile time. The stack section plays a very important role in the memory because whenever the function is called, a new stack frame is created.

Stack is also used for recursive functions. When the function is called itself again and again inside the same function which causes the stack overflow condition and it leads to the segmentation fault error in the program.

4. Heap

Heap memory is used for the dynamic memory allocation. Heap memory begins from the end of the uninitialized data segment and grows upwards to the higher addresses. The malloc() and calloc() functions are used to allocate the memory in the heap. The heap memory can be used by all the shared libraries and dynamically loaded modules. The free() function is used to deallocate the memory from the heap.

#include<stdio.h>

int main()

{

int \*ptr = (int\*)malloc(sizeof(int)) ; // memory gets allocated in the heap segment.

return 0;

}