Memory Segmentation

A la hora de ejecución de programas compilados se reserva una porción de memoria, la cual posteriormente se divide en 5 categorias diferentes:

- ✗ Text Segment (Segmento de código) → Es la poción de memoria encargada de alojar el **código compilado**, es decir, las instrucciones en ensamblados que serán enviadas al procesador para la correcta ejecución del programa. Esta parte de la memoria es de solo lectura, teniendo en cuenta, que se pueden ejecutar varias instancias del programa sin que exista problemas entre ellas (ya que cada una se estará ejecutando en zonas de memoria con permisos de sólo lectura).

- ✗ Data segment(Segmento de datos) → Se encarga de almacenar las **variables globales y las variables estáticas** (constantes)

- ✗ Bss Segment → Almacena las variables no inicializadas.

Tanto el Data Segment como el Bss Segment tiene permisos de escritura, ya que como se puede observar facilmente las variables globales persisten independientemente del proceso de ejecución.

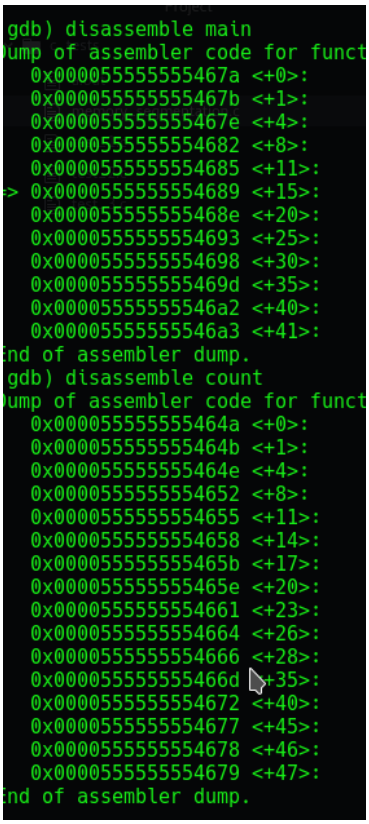
- ✗ Heap Segment(Montículo) → Es el segmento de memoria que puede ser controlado por el programador de forma directa. Este segmento de memoria puede ser redefinido, es decir, puede adaptarse a las necesidades del programador. **Parte libre de la memoria**

- ✗ Stack Segment(Pila) → Es un segmento con tamaño variable y es la parte de memoria encargada de almacenar las variables temporales durante las llamadas a las funciones (es como se muestra el comando *bt* en gdb). **Variables locales e información de control.**

El **stack** es una estructura de programación abstracta, que sigue una ordenación FILO (First Input, Last Output) esto significa que el primer elemento que colocamos en el stack es el último elemento en salir del mismo. Cuando un elemento es colocado en el stack es conocido como *pushing* y cuando es eliminado es *popping.*

Stack Segment (El stack crece hacia direcciones bajas) Heap Segment (e*l heap crece hacia direcciones altas)*
Bss segment
Data Segment
Text(code) segment

memory_segmentation.c



Ejecutando el código en el debugger gdb y haciendo un desensamblado de las funciones (para obtener las direcciones de memoria de las variables de las mismas) podemos comprobar con facilidad que las instrucciones (así como las variables) de la función *main* son más altas que las direcciones de memoria de la función *count.*

¿Qué podemos sacar en conclusión de esto? La respuesta es fácil, simplemente tenemos que observar que las variables locales usadas en *count* estan dentro del stack(pila), puesto que crecen hacia abajo y la función *main* esta en el heap(montículo), puesto que las direcciones de memoria son crecientes.

La función count esta por debajo de la memoria que la funcion main.

Podemos comprobar lo mismo, con un sencillo código en c

memory_examples.c

malloc_example.c | memory_segmentation.c | memory_examples.c

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

int global_var;
int global_var_inicializada = 5;

void function(){

  int stack_var;
  printf("Direccion de memoria de variable en stack %p\n", &stack_var);
}

int main(int argc, char const *argv[]) {
  int stack_var;
  static int static_var;
  static int static_var_inicializada = 2;
  int *heap_puntero;

  heap_puntero = (int *) malloc(sizeof(int));

  printf("Data Segment\n");
  printf("%p\n%n",&global_var_inicializada,&static_var_inicializada );

  printf("BSS Segment\n");
  printf("%p\n%p\n",&static_var,&global_var );

  printf("Heap Segment\n" );
  printf("%p\n",&heap_puntero );

  printf("Stack Segment\n" );
  printf("%p\n",&stack_var );

  function();

  return 0;
}
```

Obteniendo las siguientes direcciones de memoria.

Trabajando con la memoria del montículo (heap)

Para trabajar con la zona de memoria asignada al montículo, tenemos que hacer uso de una serie de funciones que nos permitirán reservar especios en la memoria según la necesitemos en nuestro programa

Funcion malloc()

La función malloc nos permite reservar una determinada cantidad de memoria dentro del heap, esta función se encarga de devolver un puntero de tipo *void* (que luego convertiremos haciendo *typecasting*), en el caso de que tenga algun fallo devolverá un puntero NULL (no apunta a ningun sitio)

malloc_example.c

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


int main(int argc, char const *argv[]) {
  int *puntero_para_heap;
  //Con sizeof nos encargamos de ver el tamaño de una determinada variable
  puntero_para_heap = (int *) malloc(sizeof(int));

  printf("%p\n",puntero_para_heap );

  *puntero_para_heap = 213213123;

  printf("%i\n",*puntero_para_heap );
  return 0;
}
```

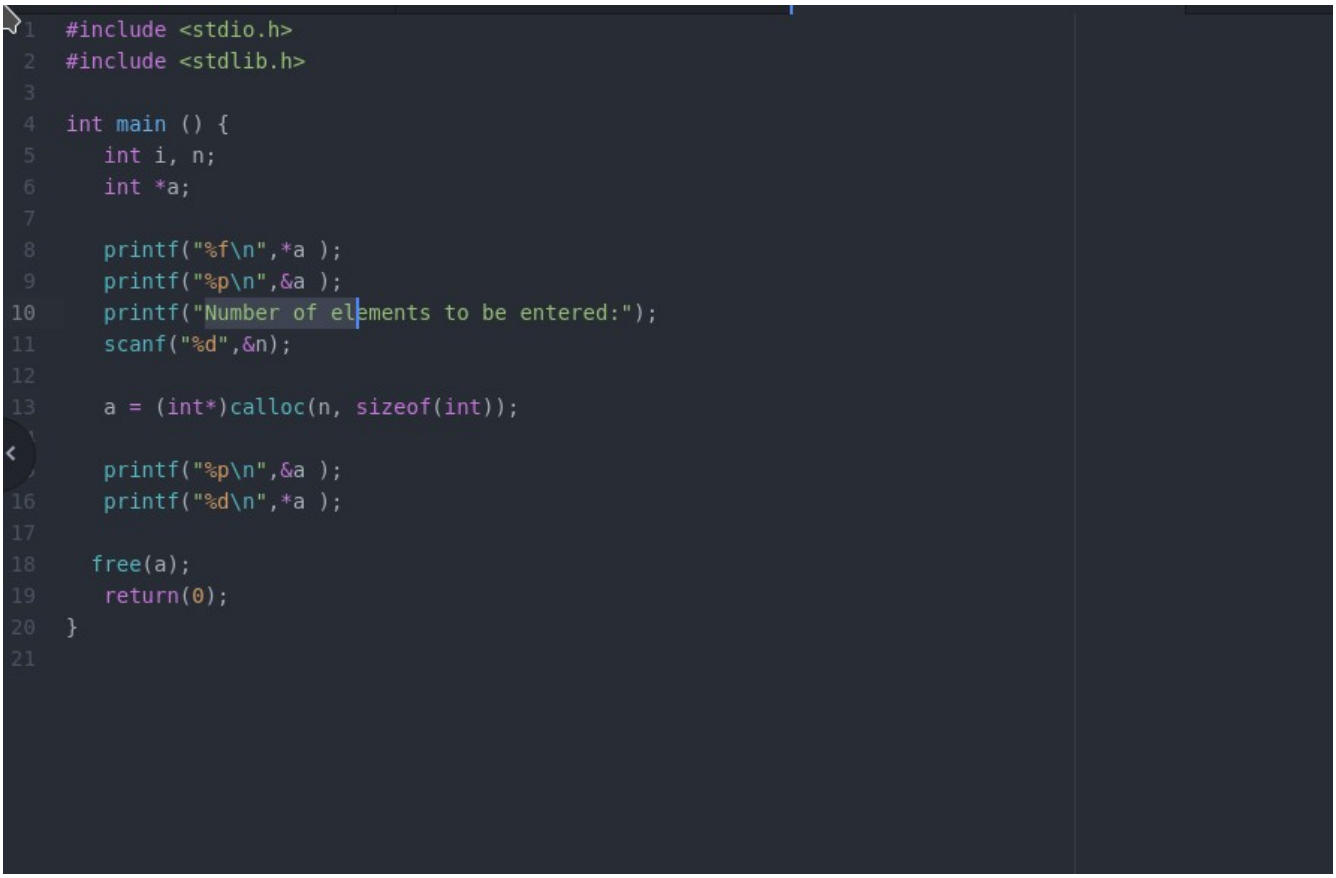
Funcion free()

Al igual que estamos reservando espcio en memoria, esta no es infinita (para desgracia del programador), existirá ocasiones donde tendrémos que estar liberando la memoria usada por nuestros programas, para ello existe la funcion free, que podemos comprobar como se implementa en el siguiente código:

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


int main(int argc, char const *argv[]) {
  int *puntero_para_heap;
  //Con sizeof nos encargamos de ver el tamaño de una determinada variable
  puntero_para_heap = (int *) malloc(sizeof(int));

  printf("%p\n",puntero_para_heap );

  *puntero_para_heap = 213213123;

  printf("%i\n",*puntero_para_heap );

  free(puntero_para_heap);
  printf("After free\n");
  printf("%i\n",*puntero_para_heap );
  return 0;
}

```

Función calloc()

Calloc es una función muy parecida a malloc, calloc se encarga de reservar un espacio en memoria e inicializarlo a cero, cosa que malloc no hace(no escribe en esa memoria). Tenemos que indicar la cantidad de elementos que queremos introducir y el tamaño de los mismos.



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

int main () {
   int i, n;
   int *a;

   printf("%f\n",*a );
   printf("%p\n",&a );
   printf("Number of elements to be entered:");
   scanf("%d",&n);

   a = (int*)calloc(n, sizeof(int));

   printf("%p\n",&a );
   printf("%d\n",*a );

  free(a);
   return(0);
}
```

Funcion realloc()

Esta función nos permite reservar un espacio en la memoria, sin embargo, la diferencia con malloc y calloc es que podemos reasignar el tamaño de este bloque con posterioridad.

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

int main () {
   char *str;


   str = (char *) malloc(15);
   strcpy(str, "cyberh99");
   printf("String = %s,  Address = %p\n", str, str);


   str = (char *) realloc(str, 25);
   strcat(str, ".com");
   printf("String = %s,  Address = %p\n", str, str);

   free(str);

   return(0);
}
```