

CURSOS
INTERSEMESTRALES



PROTECO

Memoria Dinamica



C Intermedio Enero 2018

Memoria dinámica

Es memoria que se reserva en **tiempo de ejecución**. Su principal ventaja frente a la estática, es que su **tamaño puede variar durante la ejecución del programa**. (En C, el programador es encargado de liberar esta memoria cuando no la utilice más). El uso de memoria dinámica es necesario cuando de antemano **no conocemos el número de datos/elementos a tratar**.



Memoria estática

Es el espacio en memoria que se crea al declarar variables de cualquier tipo de dato (primitivas [int,char...] o compuestas[estructuras, matrices, apuntadores...]). **La memoria que estas variables ocupan no puede cambiarse durante la ejecución y tampoco puede ser liberada manualmente.**



Segmentos de Memoria

1-Segmento de código.

Espacio en el cual se guarda todo nuestro código.

2-Segmento de datos.

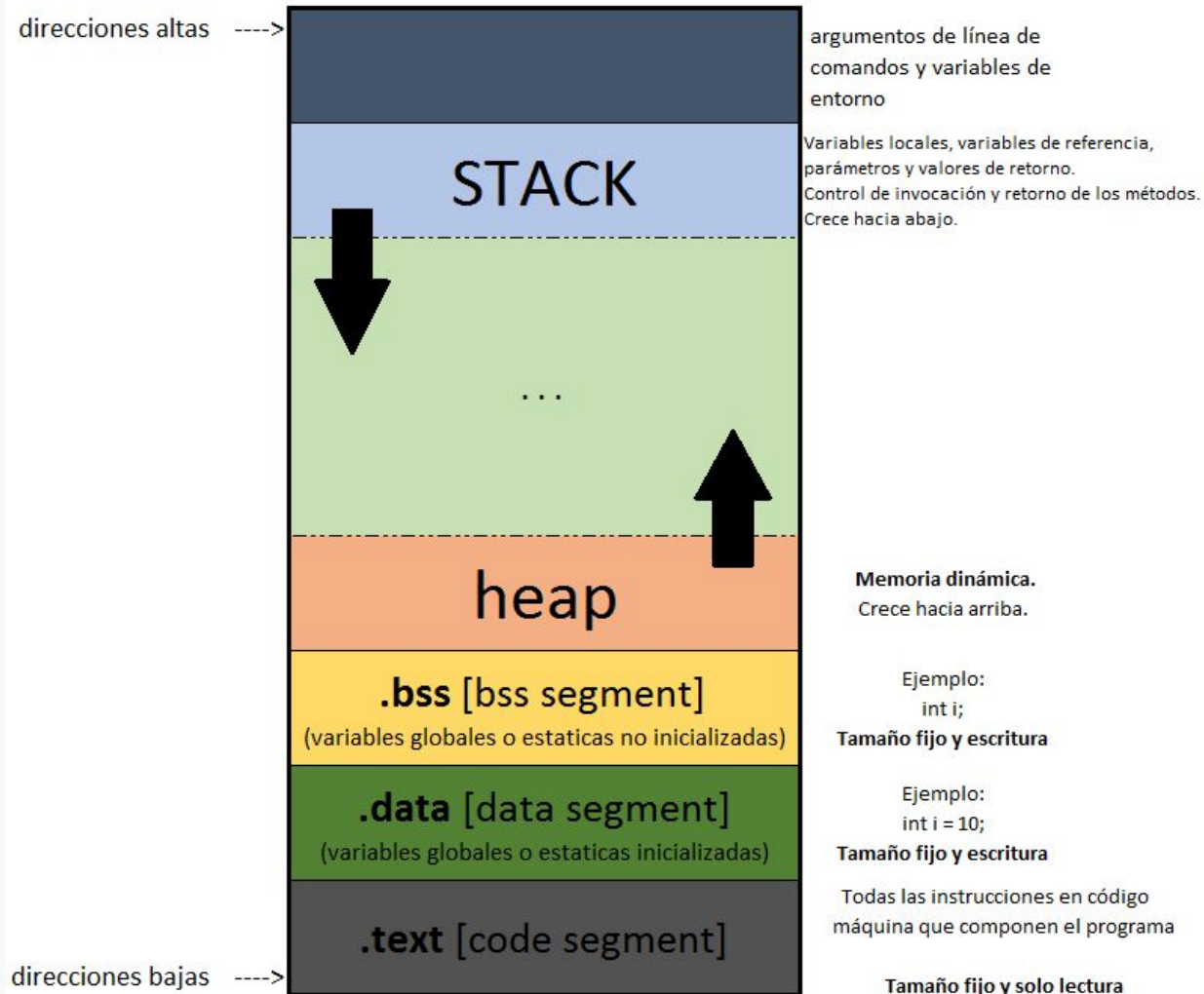
Almacena las variables que declaramos en nuestro código.

3-Segmento de memoria.

Es utilizado por las funciones de nuestro código.



Segmentos de Memoria



Heap

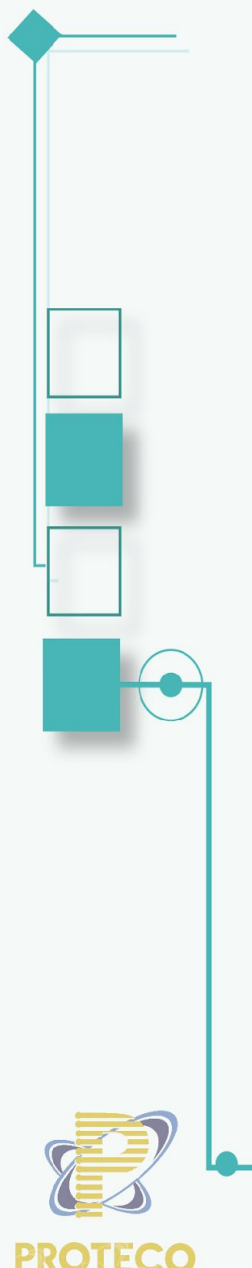
Segmento de memoria reservado para la memoria dinámica del programa. Crece hacia a arriba, en el mismo sentido que las direcciones de memoria. **Para reservar memoria utilizamos, por ejemplo, las conocidas funciones de asignación `malloc()`, `calloc()`, o `realloc()` y para liberar la memoria es `free()` del lenguaje C.**



Algunas consideraciones y aspectos sobre el Heap:

- Su tamaño se ve únicamente limitado por el tamaño de la memoria RAM.
- La manipulación del Heap (asignación, lectura, escritura) es mas lenta que la del Stack .
- Esta memoria se mantiene en uso hasta que se libera explícitamente por el programa. (o es liberada por el SO al terminar la ejecución del mismo)
- Puede ser accedida desde fuera del bloque donde fue asignada.
- El tamaño de este segmento no está predefinido, va variando dependiendo de la ejecución de programa.





```
void liberar();  
int *x;  
void main(){  
    if(true) {  
        x = malloc(sizeof(int));  
    }  
    *x = 1;  
    liberar();  
}  
void liberar(){  
    free(x);  
}
```



Stack

La pila o stack. Aquí se guardan los argumentos pasados al programa, las cadenas del entorno donde este es ejecutado, argumentos pasados a las funciones, las variables locales que todavía no contienen ningún contenido.

El modo en como se asignara la memoria en el Stack, se define durante el **proceso de compilado**.



Algunas consideraciones y aspectos sobre el Stack:

- Las variables almacenadas en el Stack son almacenadas directamente a esta memoria. Estructura LIFO.
- Su acceso es muy rápido.
- Es liberada al terminar la ejecución.
- Fragmentos grandes de memoria, como arrays de gran envergadura, no deberían ser almacenados en el Stack, para prevenir desbordamientos (Stack Overflow).
- Las variables almacenadas en el Stack solamente son accesibles desde el bloque de código donde fueron declaradas.

```
void main() {  
    if(true) {  
        int x = 0;  
    }  
    x = 1;  
}
```



Stacks Vs Heap

Heap

- Variables pueden ser accesibles globalmente
- No tiene limite en memoria
- Acceso mas lento que el Stack
- Al utilizar memoria del Heap es necesario liberarla.
- Nosotros manejamos la memoria
- Variables pueden ser redimensionadas con `realloc()`

Stack

- Acceso más rapido
- No especificamos como guardarla
- Espacio manejado eficientemente por el CPU.
- La memoria no puede ser fragmentada.
- Solo variables locales
- Limite en el tamaño del Stack depende del OS.
- variables no pueden ser redimensionadas.



Sizeof

Sizeof

El operador **sizeof(operando)** entrega la cantidad de almacenamiento en bytes que ocupa el operando.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      printf("Tamaño de los diferentes tipos de datos:\n\n");
7      printf("    char: %2d\n", sizeof(char));
8      printf("  short int: %2d\n", sizeof(short int));
9      printf("    int: %2d\n", sizeof(int));
10     printf("  long int: %2d\n", sizeof(long));
11     printf("    float: %2d\n", sizeof(float));
12     printf("    double: %2d\n", sizeof(double));
13     printf(" long double: %2d\n", sizeof(long double));
14     printf("    pointer: %2d\n", sizeof(void *));
15     return 0;
16 }
```



Malloc

La función malloc reserva un bloque de memoria y devuelve un **apuntador void *** al inicio de la misma. **void *malloc(size_t size);**

Donde el parámetro size especifica el número de bytes a reservar. En caso de que no se pueda realizar la asignación, devuelve el valor nulo (definido en la macro NULL), lo que permite saber si hubo errores en la asignación de memoria.



Malloc

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

```
int main(){
```

```
    int *i;
```

/*Reservamos memoria suficiente para almacenar un int y asignamos su dirección a i*/

```
    i=(int*)malloc(sizeof(int));
```

```
    if(i == NULL){
```

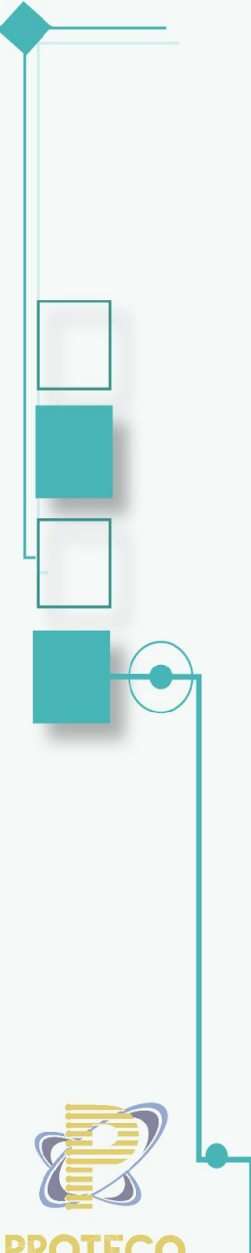
/*Error al intentar reservar memoria*/

```
    }
```

```
    return 0;
```

```
}
```





```
#include <stdio.h>
int main(){
    int *arr, n;
    printf("Numero de elementos del arreglo\n");
    scanf("%d",&n);
    arr =(int*)malloc (n * sizeof(int));
    if(arr == NULL){
        /*Error al intentar reservar memoria*/
    }
    return 0;
}
```



Calloc

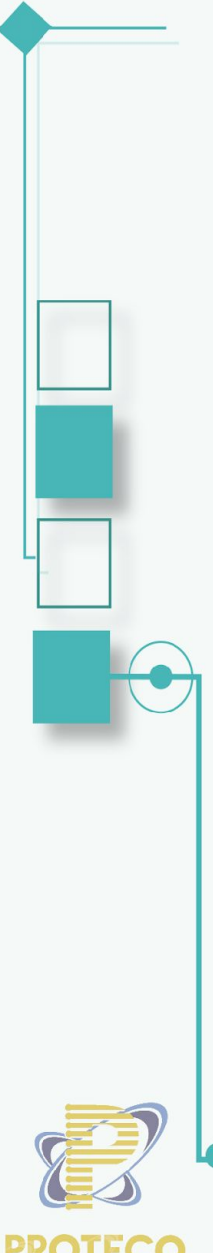
La función calloc funciona de modo similar a malloc, pero además de reservar memoria, inicializa a 0 la memoria reservada.

Se usa comúnmente para arreglos y matrices. Está definida de esta forma:

```
void *calloc(size_t x, size_t size);
```

El parámetro x indica el número de elementos a reservar, y size el tamaño de cada elemento.





```
#include <stdio.h>
int main(){
    int *arr, n;
    printf("Numero de elementos del arreglo\n");
    scanf("%d",&n);
    arr = (int*)calloc (n,sizeof(int));
    if(arr == NULL){
        /*Error al intentar reservar memoria*/
    }
    return 0;
}
```



Realloc

La función realloc redimensiona el espacio asignado de forma dinámica anteriormente a un apuntador.

void *realloc(void *ptr, size_t size);

Donde **ptr** es el apuntador a redimensionar, y **size** el nuevo tamaño, en bytes, que tendrá.

Si el apuntador que se le pasa tiene el valor nulo, esta función actúa como malloc.

Si la reasignación no se pudo hacer con éxito, devuelve un apuntador nulo, dejando intacto el apuntador que se pasa por parámetro.

Al usar realloc, se debería usar un apuntador temporal. De lo contrario, podríamos tener una fuga de memoria, si es que ocurriera un error en realloc.



```
/* Reservamos 5 bytes */  
void *ptr = malloc(5);  
...  
/* Redimensionamos el puntero (a 10 bytes) y lo asignamos a un puntero temporal */  
void *tmp_ptr = realloc(ptr, 10);  
  
if (tmp_ptr == NULL) {  
    /* Error: tomar medidas necesarias */  
}  
else {  
    /* Reasignación exitosa. Asignar memoria a ptr */  
    ptr = tmp_ptr;  
}
```



Free

La función free sirve para liberar memoria que se asigna dinámicamente.

Si el apuntador es nulo, free no hace nada.

void free(void *ptr);

El parámetro ptr es el apuntador a la memoria que se desea liberar.

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

```
int main(){  
    int *i;  
    i=(int*)malloc(sizeof(int));  
    if(i == NULL){  
    }  
    free(i);  
    i=NULL;  
    return 0;  
}
```



Ejercicios:

1. Reservar Memoria para 4 tipos de Datos
2. Reservar con el Stack
3. Reservar con malloc para un arreglo de 10 elementos
4. Reservar con Calloc para un Arreglo indefinido
5. Redimensionar un arreglo

