Intro a C

Memoria

With 💜 by @vichoeq & @KnowYourselves

Detalles del STACK

Si bien el STACK es muy útil, tiene algunos detalles y limitaciones.

Generación de arreglos

```c
int* seq(int from, int count)
{
  int S[count];
  for(int i = 0; i < count; i++)
  {
    S[i] = from + i;
  }
  return S;
}


int main()
{
  int* seq_7_3 = seq(7,3);
  int* seq_2_4 = seq(2,4);
```

Generación de arreglos

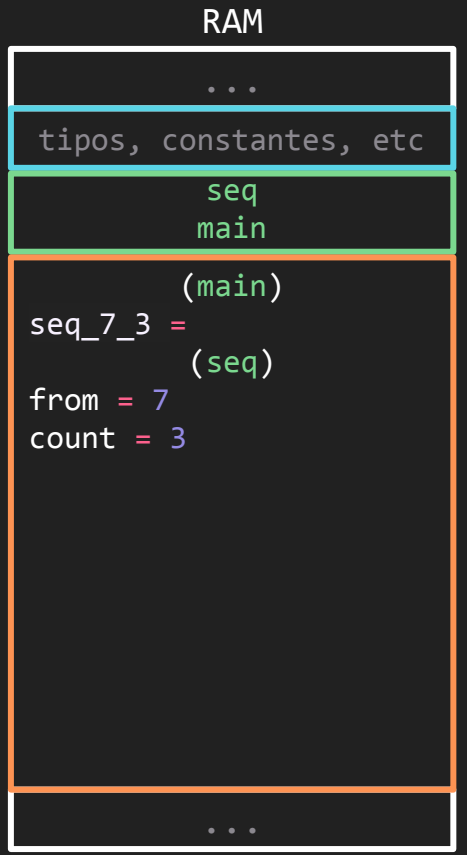
```c
int* seq(int from, int count)
{
  int S[count];
  for(int i = 0; i < count; i++)
  {
    S[i] = from + i;
  }
  return S;
}


int main()
{
  int* seq_7_3 = seq(7,3);
  int* seq_2_4 = seq(2,4);
```

Generación de arreglos

```c
int* seq(int from, int count)
{
  int S[count];
  for(int i = 0; i < count; i++)
  {
    S[i] = from + i;
  }
  return S;
}


int main()
{
  int* seq_7_3 = seq(7,3);
  int* seq_2_4 = seq(2,4);
```

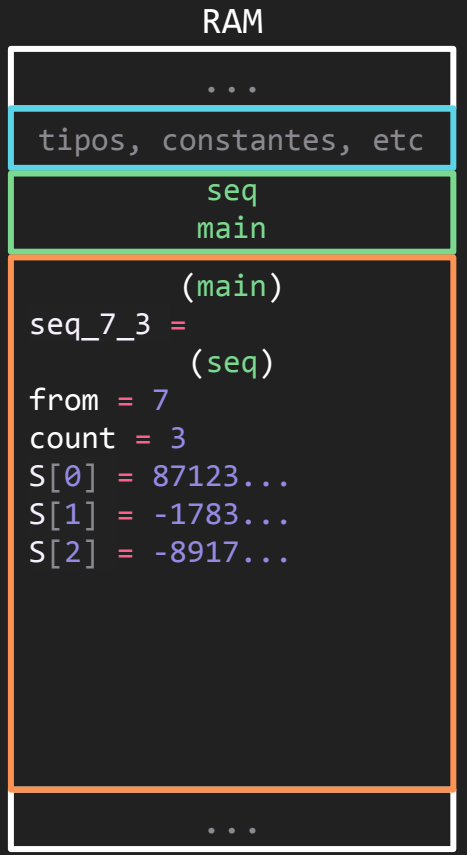


Generación de arreglos

```c
int* seq(int from, int count)
{
  int S[count];
  for(int i = 0; i < count; i++)
  {
    S[i] = from + i;
  }
  return S;
}


int main()
{
  int* seq_7_3 = seq(7,3);
  int* seq_2_4 = seq(2,4);
```

0x8a71f0 →

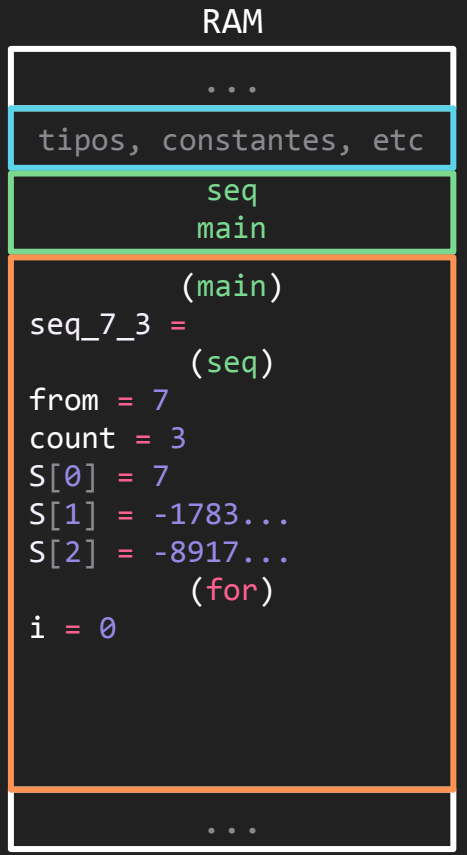


Generación de arreglos

```c
int* seq(int from, int count)
{
  int S[count];
  for(int i = 0; i < count; i++)
  {
    S[i] = from + i;
  }
  return S;
}


int main()
{
  int* seq_7_3 = seq(7,3);
  int* seq_2_4 = seq(2,4);
```

0x8a71f0 →

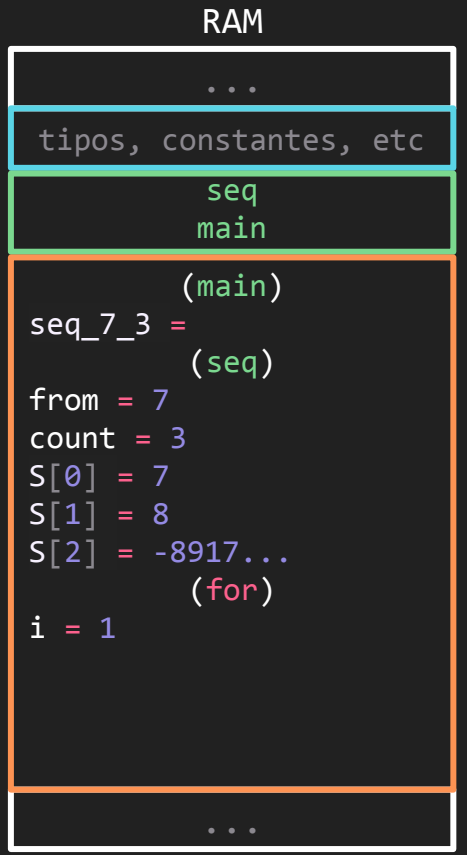


Generación de arreglos

```c
int* seq(int from, int count)
{
  int S[count];
  for(int i = 0; i < count; i++)
  {
    S[i] = from + i;
  }
  return S;
}


int main()
{
  int* seq_7_3 = seq(7,3);
  int* seq_2_4 = seq(2,4);
```

0x8a71f0 →

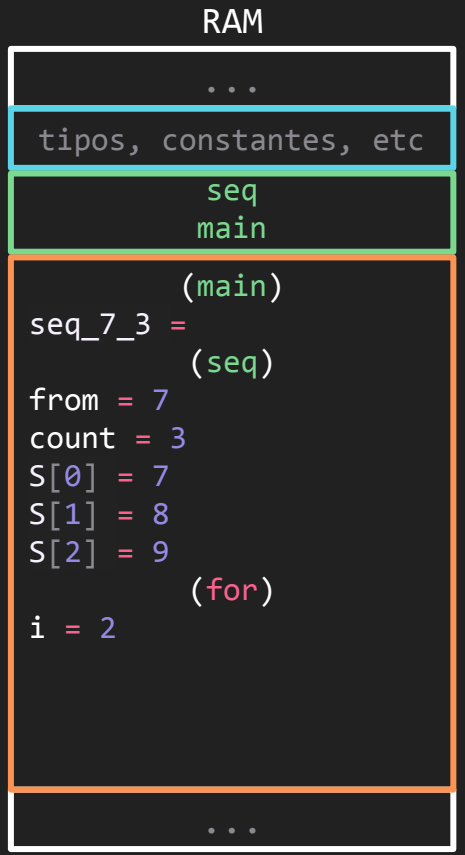


Generación de arreglos

```c
int* seq(int from, int count)
{
  int S[count];
  for(int i = 0; i < count; i++)
  {
    S[i] = from + i;
  }
  return S;
}


int main()
{
  int* seq_7_3 = seq(7,3);
  int* seq_2_4 = seq(2,4);
```

0x8a71f0 →

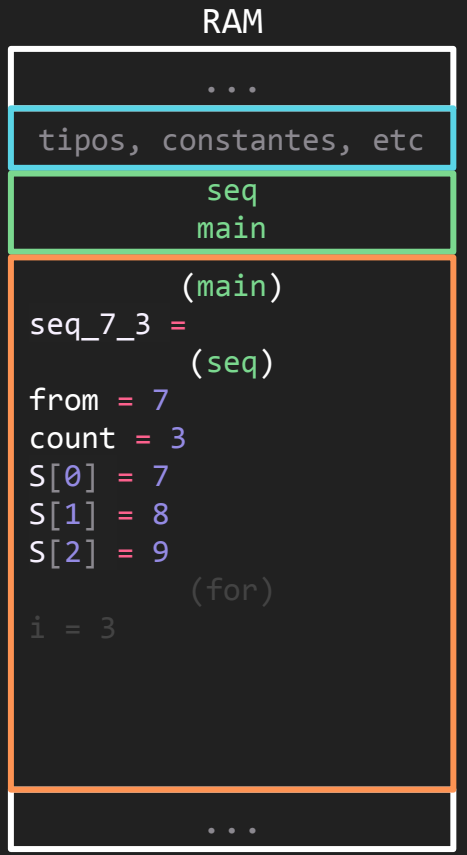


Generación de arreglos

```c
int* seq(int from, int count)
{
  int S[count];
  for(int i = 0; i < count; i++)
  {
    S[i] = from + i;
  }
  return S;
}


int main()
{
  int* seq_7_3 = seq(7,3);
  int* seq_2_4 = seq(2,4);
```

0x8a71f0 →

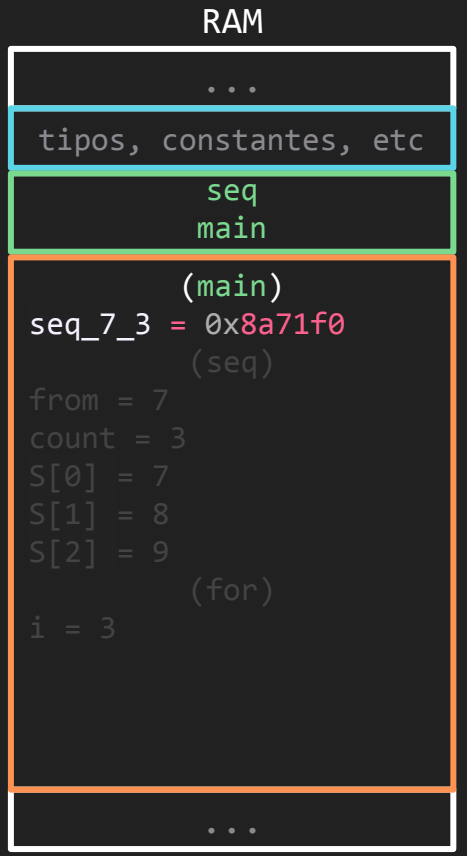


Generación de arreglos

```c
int* seq(int from, int count)
{
  int S[count];
  for(int i = 0; i < count; i++)
  {
    S[i] = from + i;
  }
  return S;
}


int main()
{
  int* seq_7_3 = seq(7,3);
  int* seq_2_4 = seq(2,4);
```

0x8a71f0 →



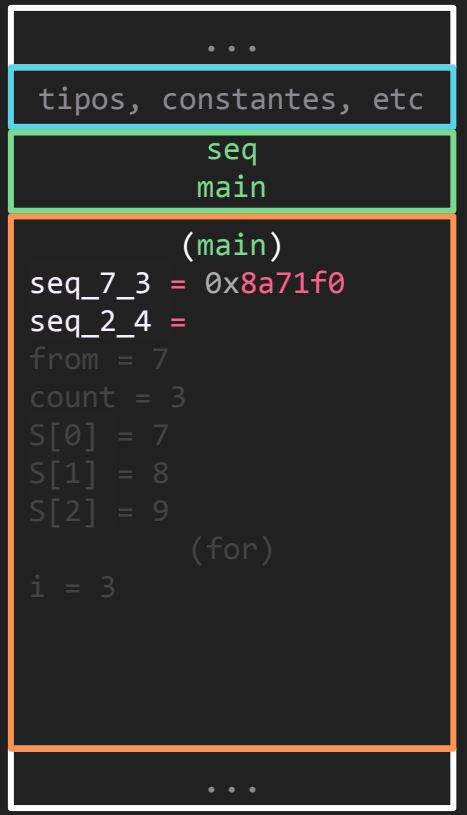
Generación de arreglos

```c
int* seq(int from, int count)
{
  int S[count];
  for(int i = 0; i < count; i++)
  {
    S[i] = from + i;
  }
  return S;
}


int main()
{
  int* seq_7_3 = seq(7,3);
  int* seq_2_4 = seq(2,4);
```

0x8a71f0 →

RAM



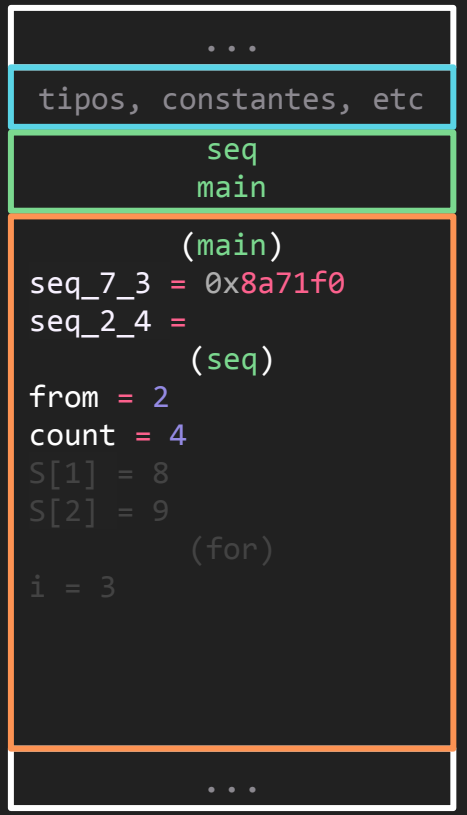
Generación de arreglos

```c
int* seq(int from, int count)
{
  int S[count];
  for(int i = 0; i < count; i++)
  {
    S[i] = from + i;
  }
  return S;
}


int main()
{
  int* seq_7_3 = seq(7,3);
  int* seq_2_4 = seq(2,4);
```

0x8a71f0 →

RAM



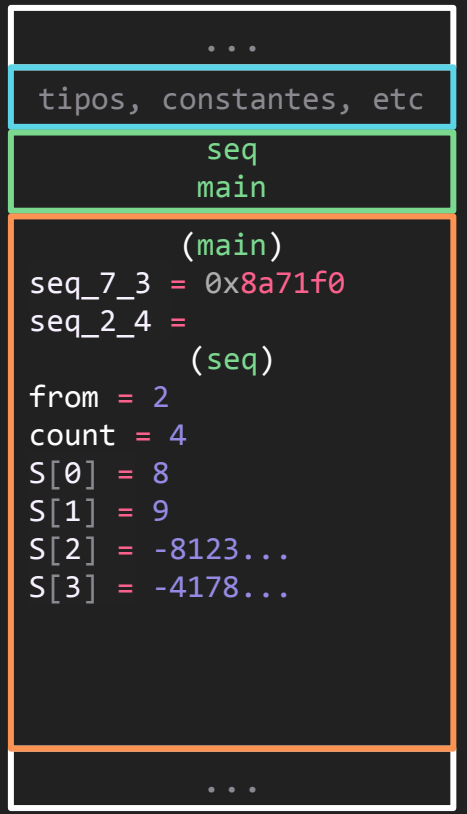
Generación de arreglos

```c
int* seq(int from, int count)
{
  int S[count];
  for(int i = 0; i < count; i++)
  {
    S[i] = from + i;
  }
  return S;
}


int main()
{
  int* seq_7_3 = seq(7,3);
  int* seq_2_4 = seq(2,4);
```

0x8a71f0 →

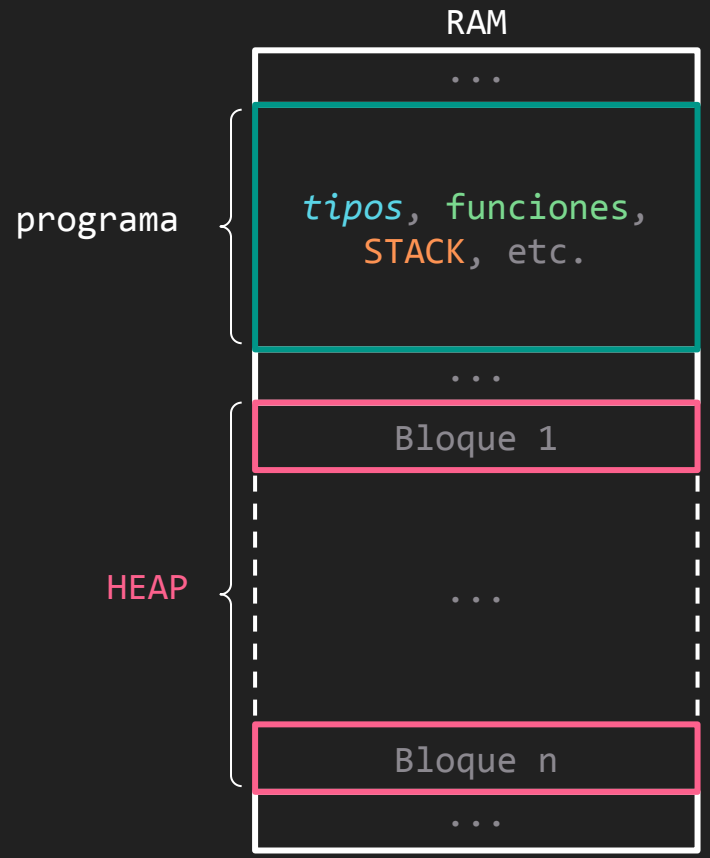
RAM



HEAP

HEAP - Definición

El HEAP es una colección de bloques de memoria solicitados por el programa.



HEAP - Características

- Sólo crece cuando el programa lo pide
- Sólo se eliminan bloques cuando el programa lo pide
- El heap no tiene límite de tamaño

Manejo de Memoria

Manejo de Memoria - <stdlib.h>

Para interactuar con el HEAP existe la librería <stdlib.h>, que nos proporciona las siguientes funciones:

- malloc
- calloc
- free

malloc - Memory Allocation

```c
int* a = malloc(sizeof(int));
printf("%p\n", a);
```

```
$ gcc main.c -o main
$ ./main
0xfa20fc
```

malloc recibe la cantidad de bytes a pedir al HEAP y retorna un puntero hacia el bloque obtenido.

malloc - Inicialización

```c
int* a = malloc(sizeof(int));
printf("%p: %d\n", a, *a);
```

```
$ gcc main.c -o main
$ ./main
0xfa20fc: -1821377856
```

El bloque al que apunta el puntero retornado por malloc no está inicializado, por lo cual podría tener cualquier valor.

malloc - Inicialización

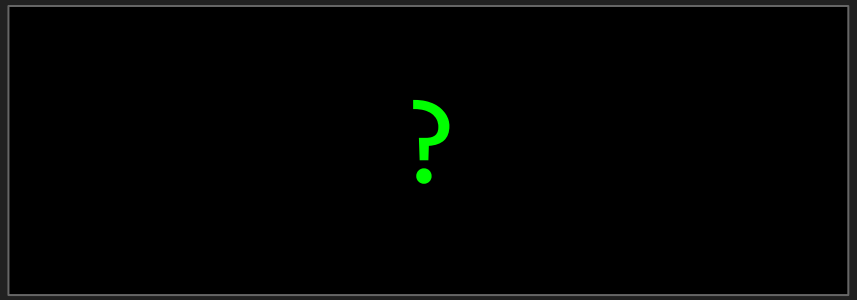
```c
int* a = malloc(sizeof(int));
*a = 5;
printf("%p: %d\n", a, *a);
```

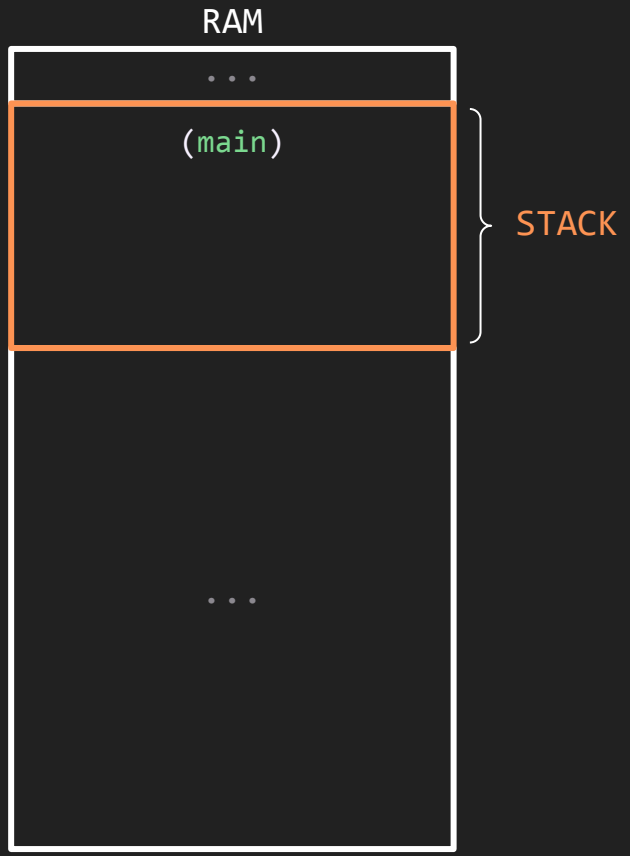
```
$ gcc main.c -o main
$ ./main
0xfa20fc: 5
```

Antes de utilizar un puntero retornado por malloc, hay que asignarle un valor.

Arreglos en el HEAP

```c
int* A = malloc(3 * sizeof(int));
A[0] = 1;
A[1] = 4;
A[2] = 3;
printf("%p\n%p\n%p\n", A, &A, &A[0]);
```

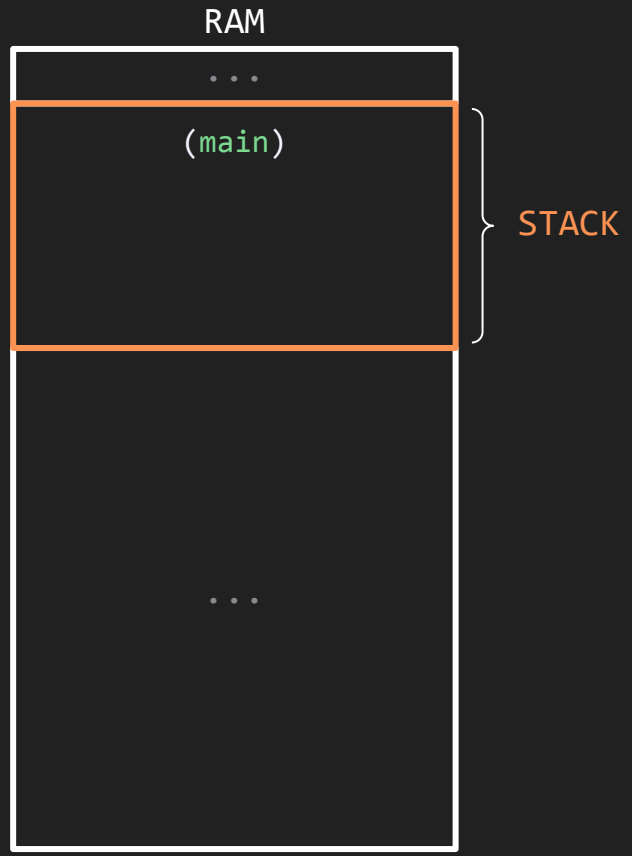




Arreglos en el HEAP

```c
int* A = malloc(3 * sizeof(int));
A[0] = 1;
A[1] = 4;
A[2] = 3;
printf("%p\n%p\n%p\n", A, &A, &A[0]);
```

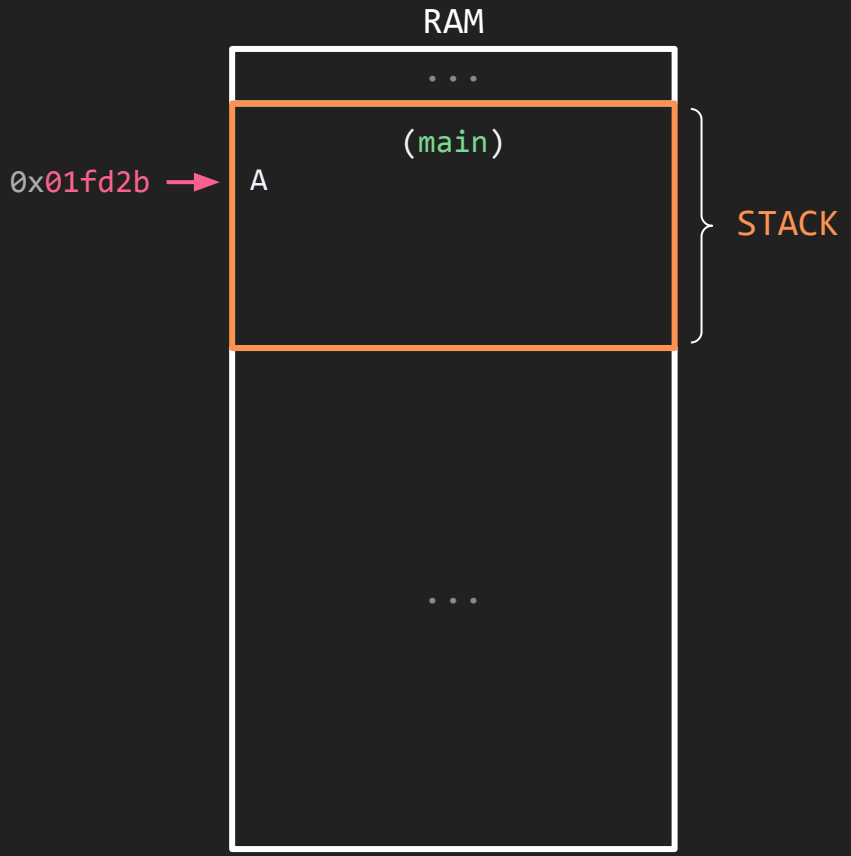
```
$ gcc main.c -o main
$ ./main
0x22d040
0x01fd2b
0x22d040
```



Arreglos en el HEAP

```c
int* A = malloc(3 * sizeof(int));
A[0] = 1;
A[1] = 4;
A[2] = 3;
printf("%p\n%p\n%p\n", A, &A, &A[0]);
```

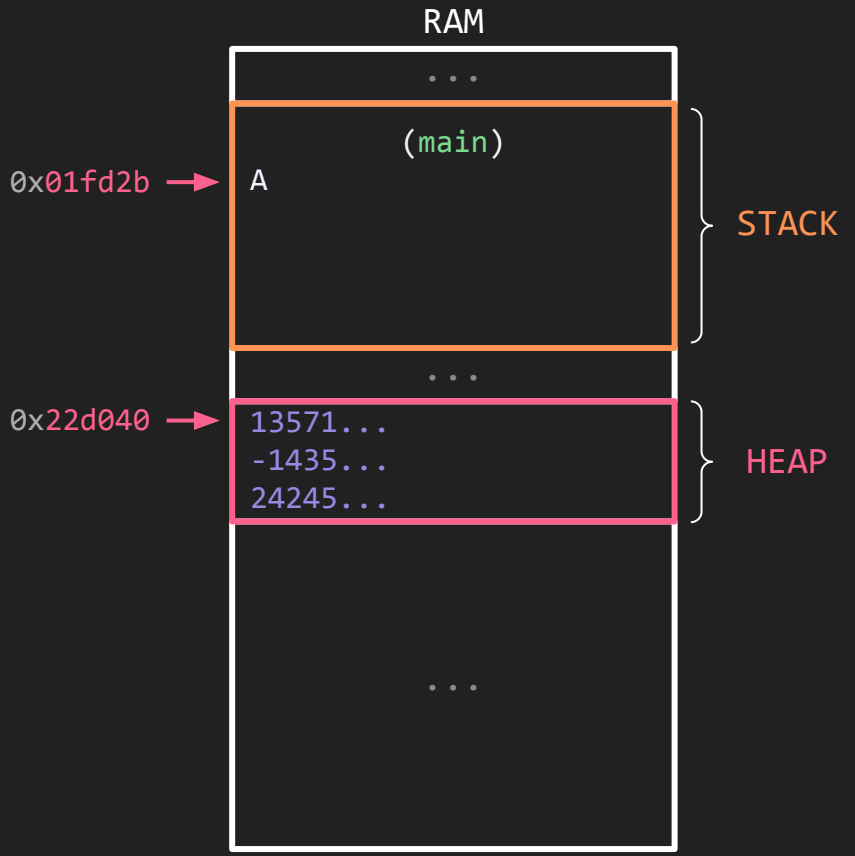
```
$ gcc main.c -o main
$ ./main
```



Arreglos en el HEAP

```c
int* A = malloc(3 * sizeof(int));
A[0] = 1;
A[1] = 4;
A[2] = 3;
printf("%p\n%p\n%p\n", A, &A, &A[0]);
```

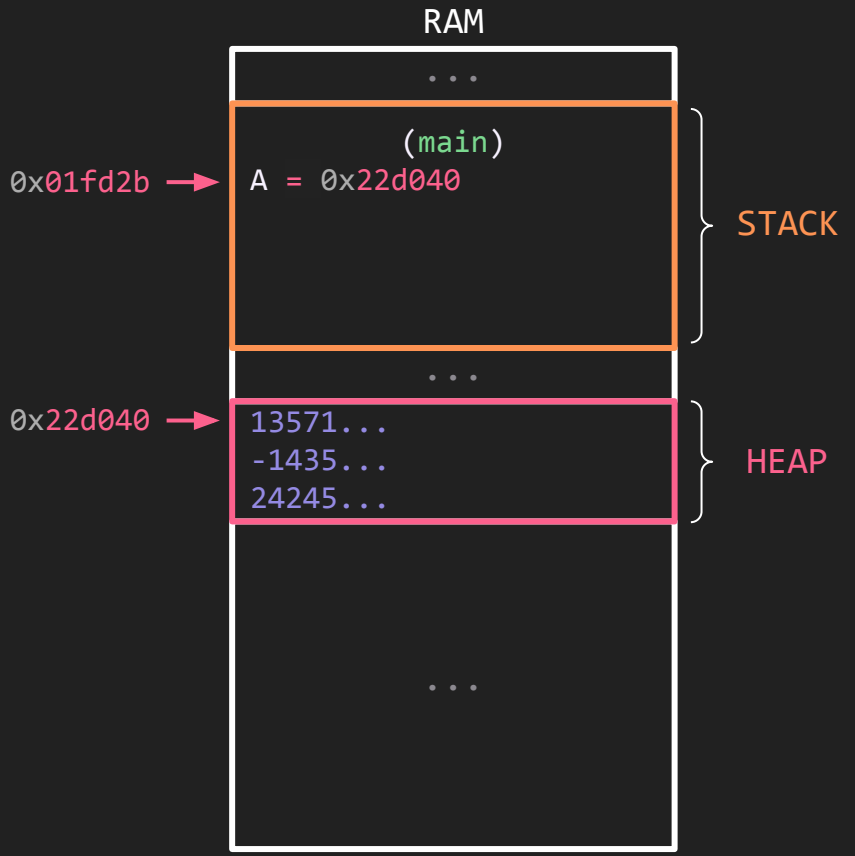
```
$ gcc main.c -o main
$ ./main
```



Arreglos en el HEAP

```c
int* A = malloc(3 * sizeof(int));
A[0] = 1;
A[1] = 4;
A[2] = 3;
printf("%p\n%p\n%p\n", A, &A, &A[0]);
```

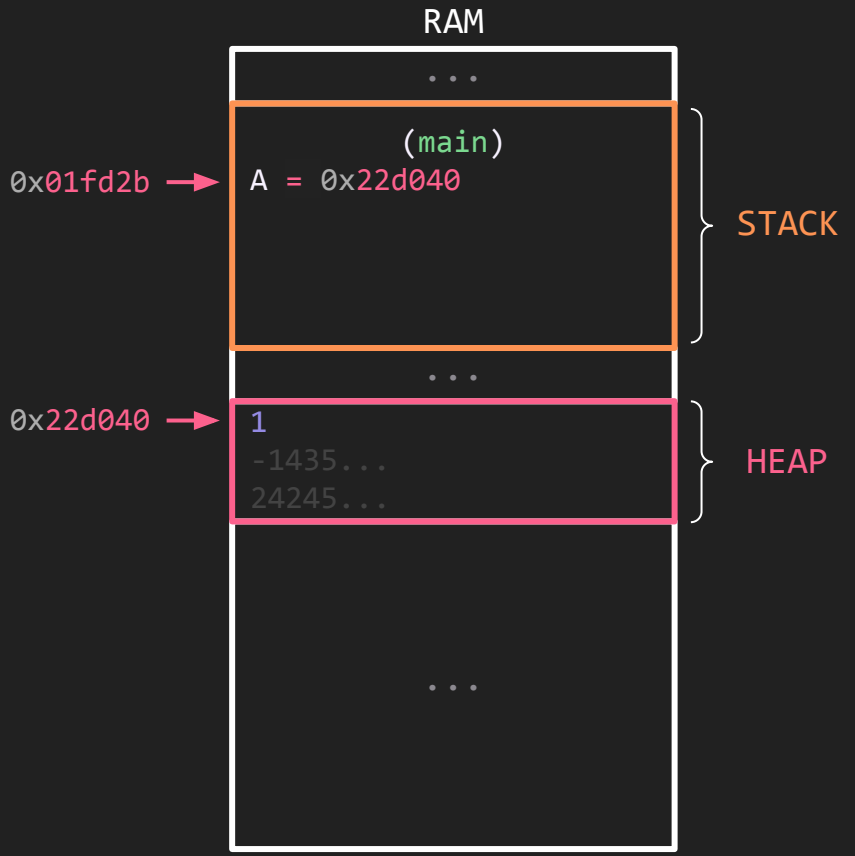
```
$ gcc main.c -o main
$ ./main
```



Arreglos en el HEAP

```c
int* A = malloc(3 * sizeof(int));
A[0] = 1;
A[1] = 4;
A[2] = 3;
printf("%p\n%p\n%p\n", A, &A, &A[0]);
```

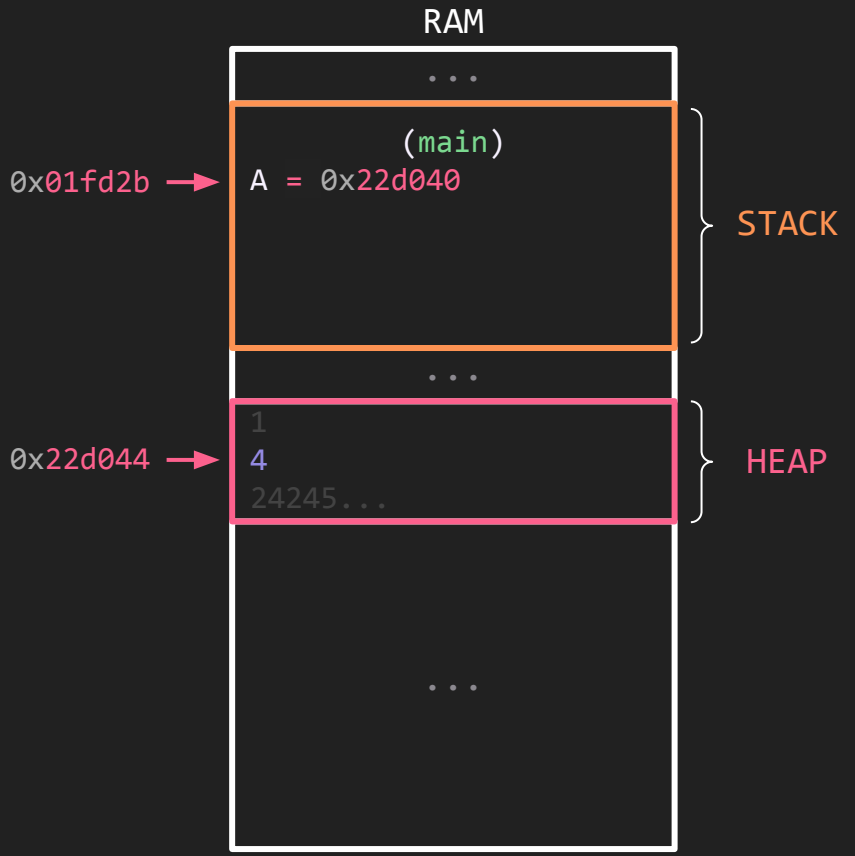
```
$ gcc main.c -o main
$ ./main
```



Arreglos en el HEAP

```c
int* A = malloc(3 * sizeof(int));
A[0] = 1;
A[1] = 4;
A[2] = 3;
printf("%p\n%p\n%p\n", A, &A, &A[0]);
```

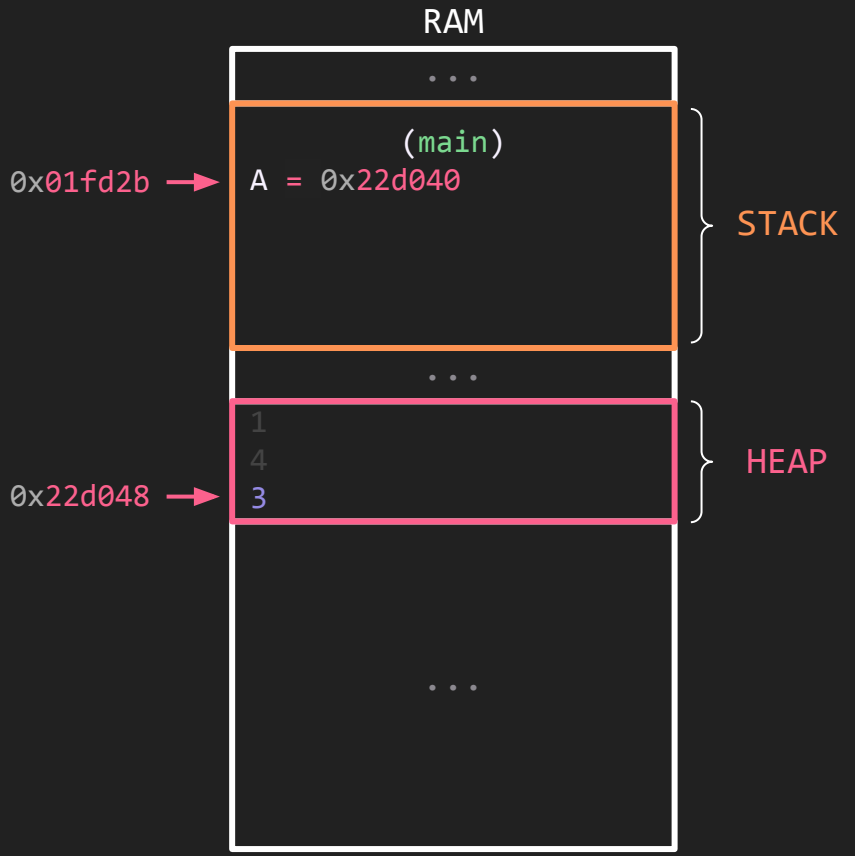
```
$ gcc main.c -o main
$ ./main
```



Arreglos en el HEAP

```c
int* A = malloc(3 * sizeof(int));
A[0] = 1;
A[1] = 4;
A[2] = 3;
printf("%p\n%p\n%p\n", A, &A, &A[0]);
```

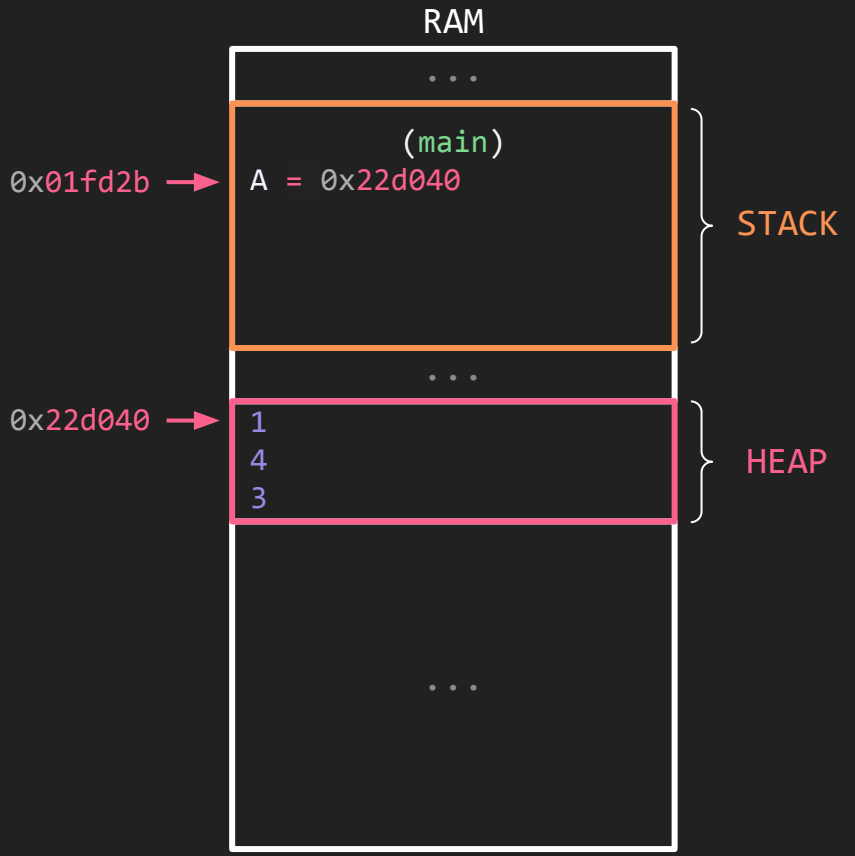
```
$ gcc main.c -o main
$ ./main
```



Arreglos en el HEAP

```c
int* A = malloc(3 * sizeof(int));
A[0] = 1;
A[1] = 4;
A[2] = 3;
printf("%p\n%p\n%p\n", A, &A, &A[0]);
```

```
$ gcc main.c -o main
$ ./main
0x22d040
0x01fd2b
0x22d040
```



calloc - Clear Memory Allocation

```c
int *a = calloc(1, sizeof(int));
printf("%p\n", a);
```

```
$ gcc main.c -o main
$ ./main
0x522d040
```

calloc recibe una cantidad de elementos y su tamaño para pedir esa cantidad de bytes al HEAP, retorna un puntero.

calloc - Inicialización

```c
int *a = calloc(1, sizeof(int));
printf("%p: %d\n", a, *a);
```

```
$ gcc main.c -o main
$ ./main
0x522d040: 0
```

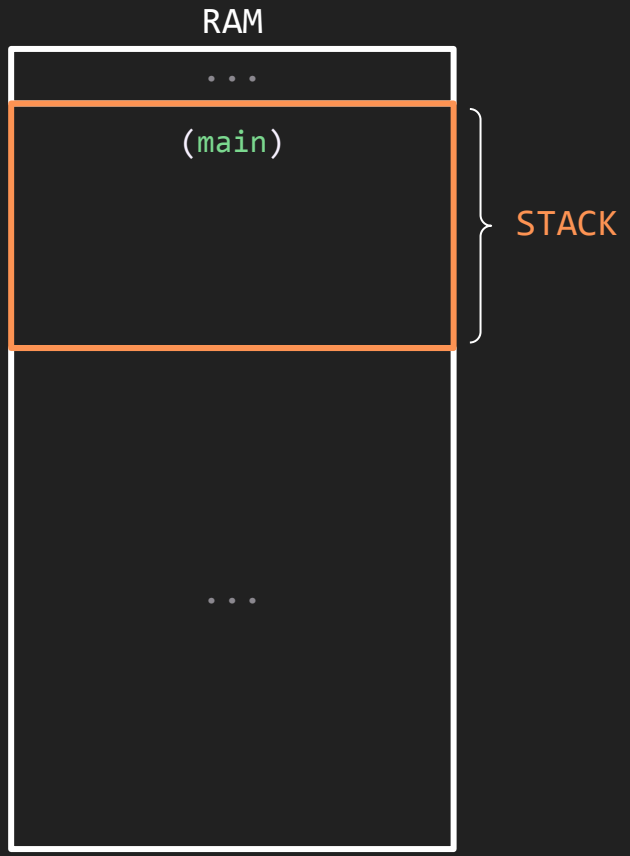
A diferencia de malloc, calloc inicializa el bloque de memoria en 0.

Cuando se quiere tener un arreglo en el HEAP, se recomienda calloc.

Arreglos en el HEAP

```c
int* A = calloc(3, sizeof(int));
A[0] = 1;
A[2] = 3;
printf("%p: %d, %d, %d\n", A, A[0], A[1], A[2]);
```

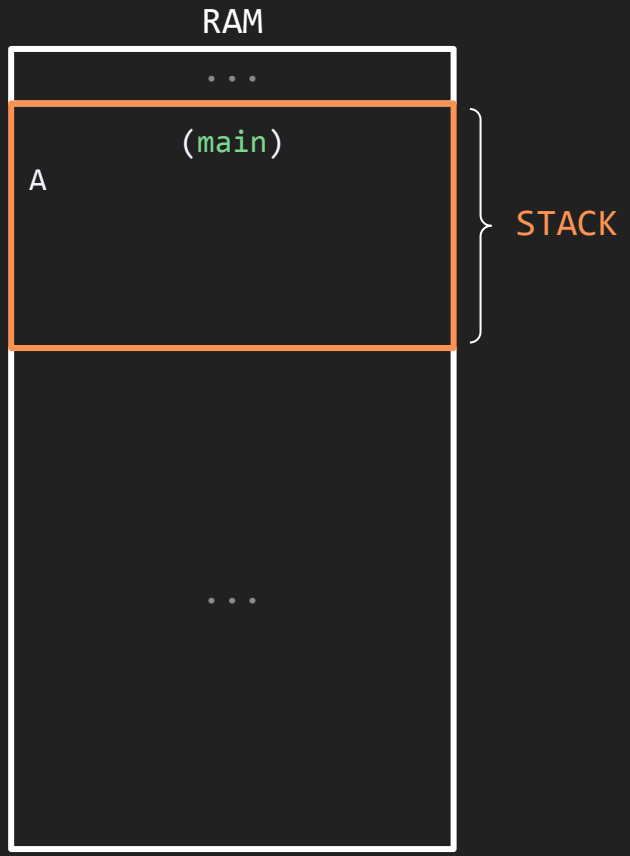
?



Arreglos en el HEAP

```c
int* A = calloc(3, sizeof(int));
A[0] = 1;
A[2] = 3;
printf("%p: %d, %d, %d\n", A, A[0], A[1], A[2]);
```

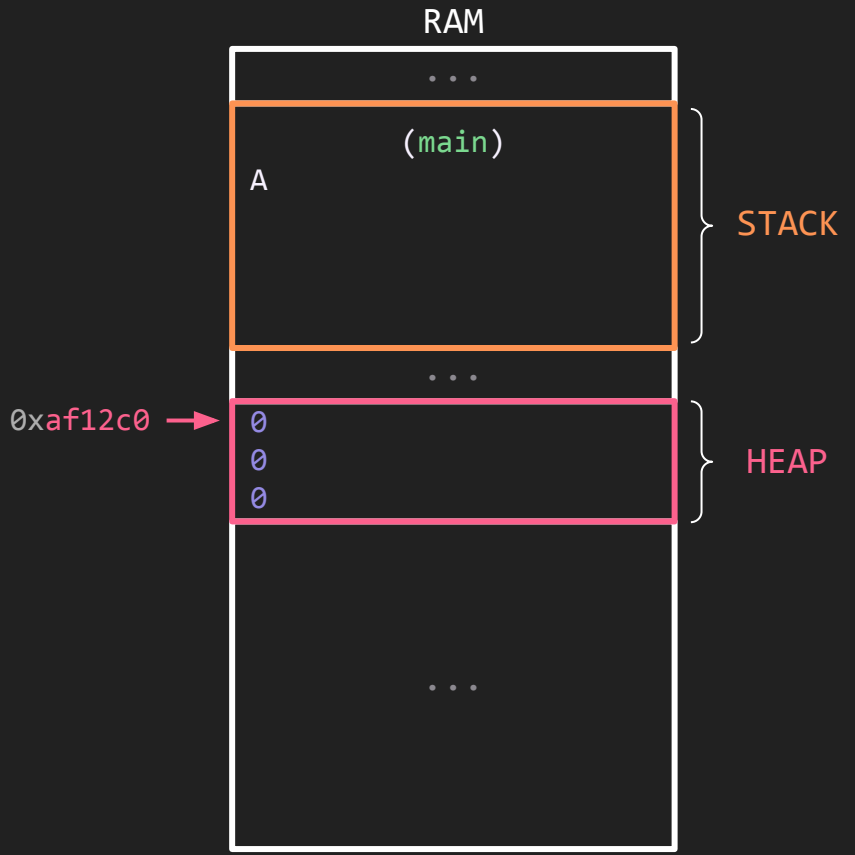
```
$ gcc main.c -o main
$ ./main
```



Arreglos en el HEAP

```c
int* A = calloc(3, sizeof(int));
A[0] = 1;
A[2] = 3;
printf("%p: %d, %d, %d\n", A, A[0], A[1], A[2]);
```

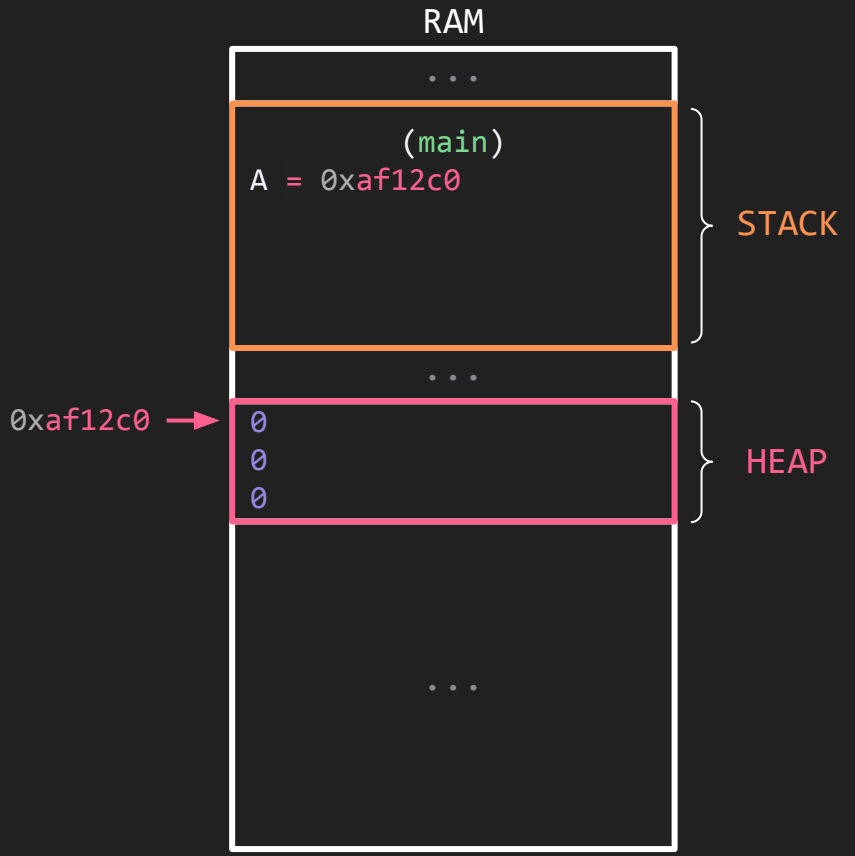
```
$ gcc main.c -o main
$ ./main
```



Arreglos en el HEAP

```c
int* A = calloc(3, sizeof(int));
A[0] = 1;
A[2] = 3;
printf("%p: %d, %d, %d\n", A, A[0], A[1], A[2]);
```

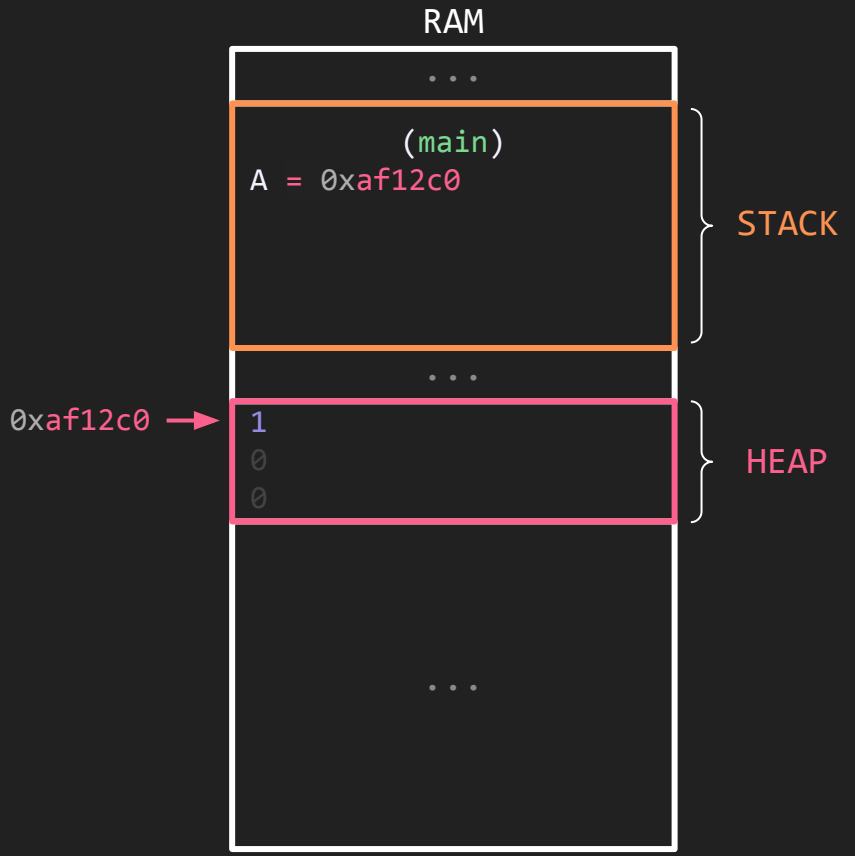
```
$ gcc main.c -o main
$ ./main
```



Arreglos en el HEAP

```c
int* A = calloc(3, sizeof(int));
A[0] = 1;
A[2] = 3;
printf("%p: %d, %d, %d\n", A, A[0], A[1], A[2]);
```

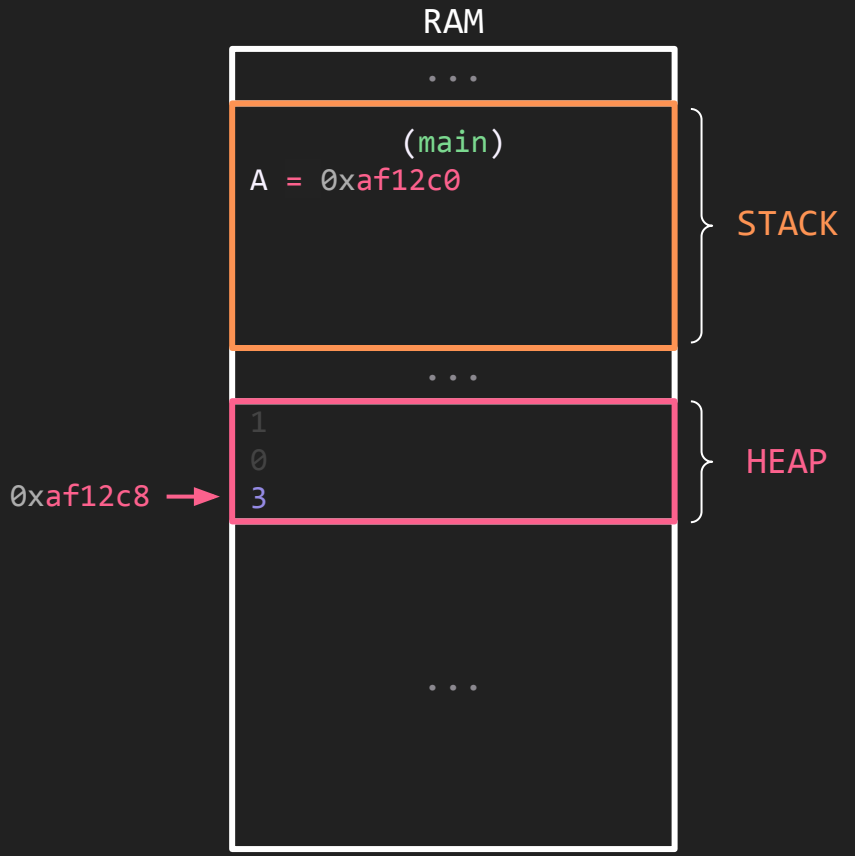
```
$ gcc main.c -o main
$ ./main
```



Arreglos en el HEAP

```c
int* A = calloc(3, sizeof(int));
A[0] = 1;
A[2] = 3;
printf("%p: %d, %d, %d\n", A, A[0], A[1], A[2]);
```

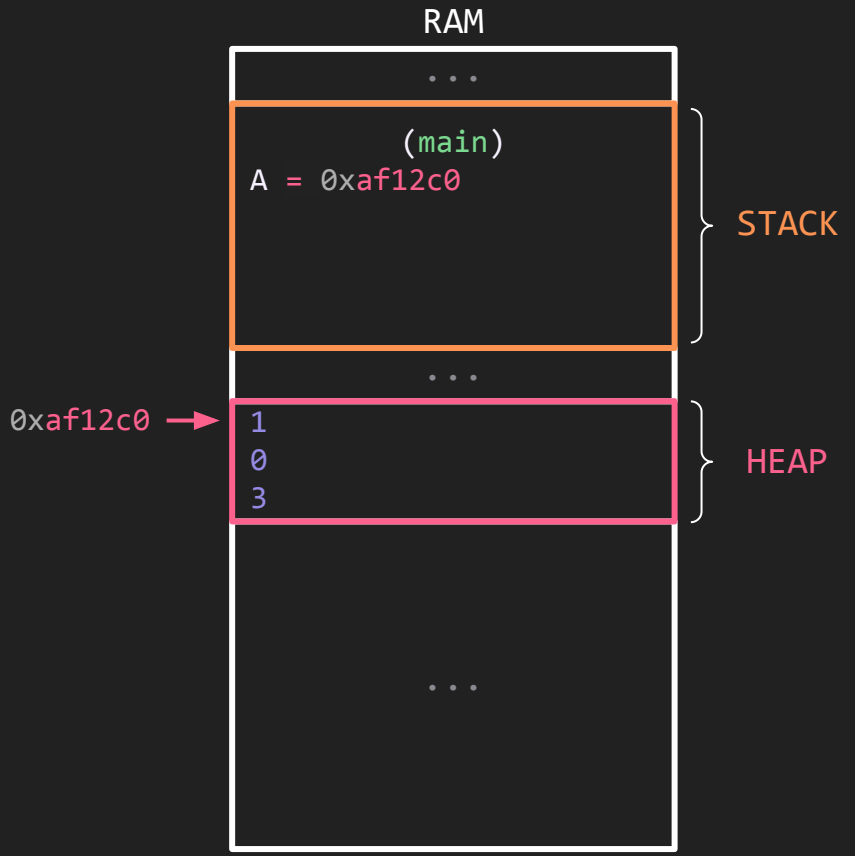
```
$ gcc main.c -o main
$ ./main
```



Arreglos en el HEAP

```c
int* A = calloc(3, sizeof(int));
A[0] = 1;
A[2] = 3;
printf("%p: %d, %d, %d\n", A, A[0], A[1], A[2]);
```

```
$ gcc main.c -o main
$ ./main
0xaf12c0: 1, 0, 3
```



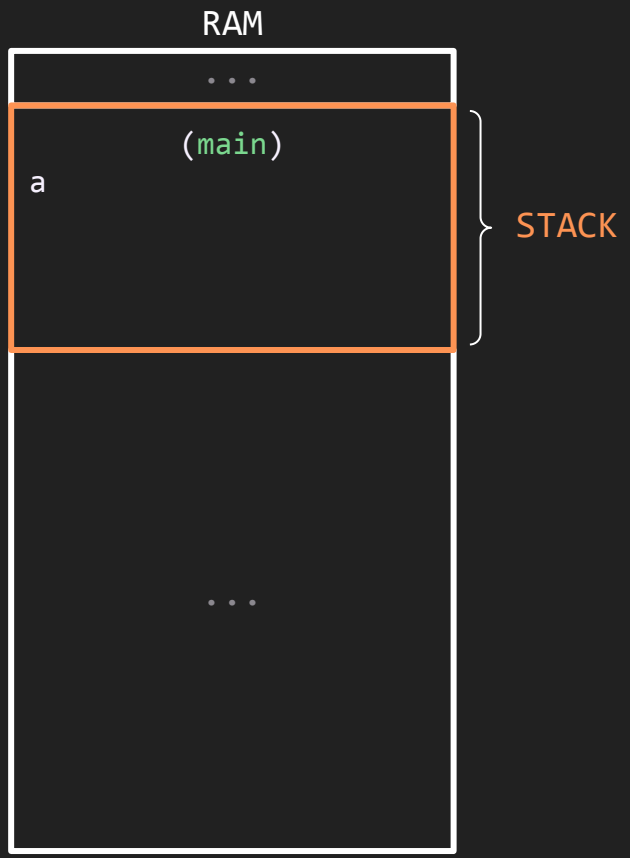
free - Memory Deallocation

```c
int* a = malloc(sizeof(int));
char *B = calloc(4, sizeof(int));
// Código
free(a);
free(B);
```

Cuando terminamos de usar un bloque del HEAP tenemos que liberarlo manualmente mediante la función free.

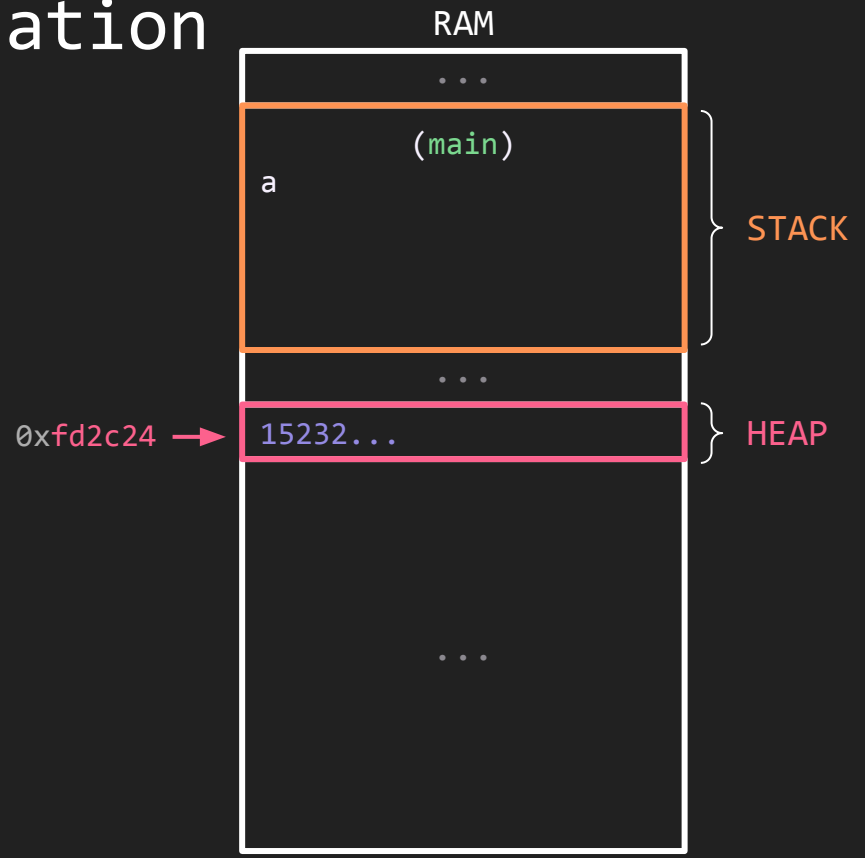
free - Memory Deallocation

```c
int* a = malloc(sizeof(int));
char *B = calloc(4, sizeof(int));
// Código
free(a);
free(B);
```



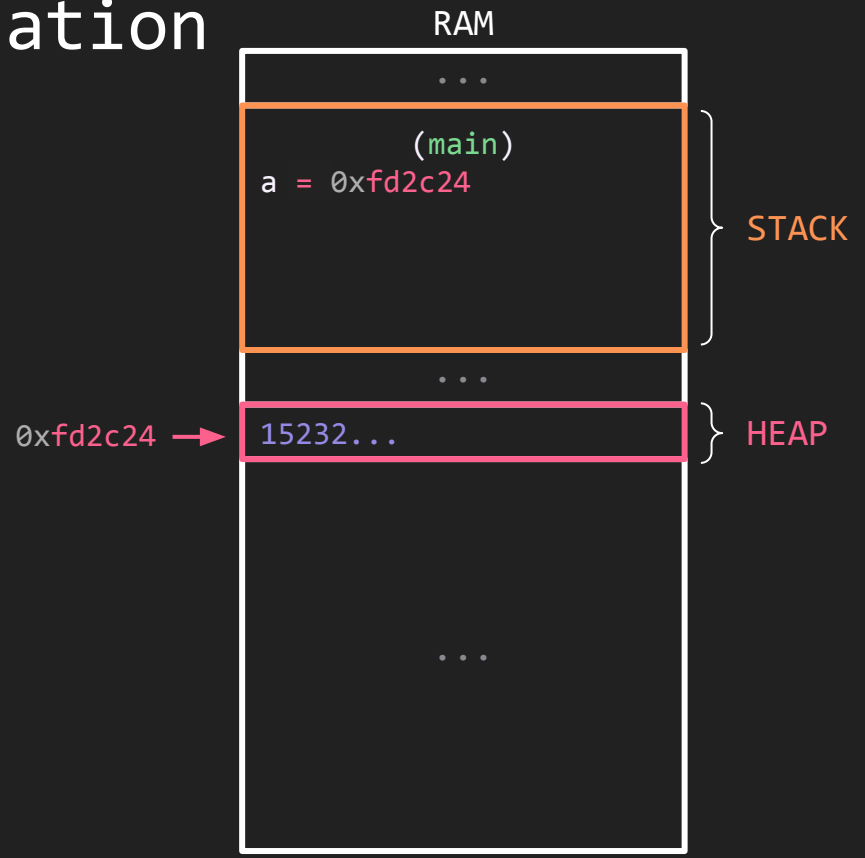
free - Memory Deallocation

```c
int* a = malloc(sizeof(int));
char *B = calloc(4, sizeof(int));
// Código
free(a);
free(B);
```



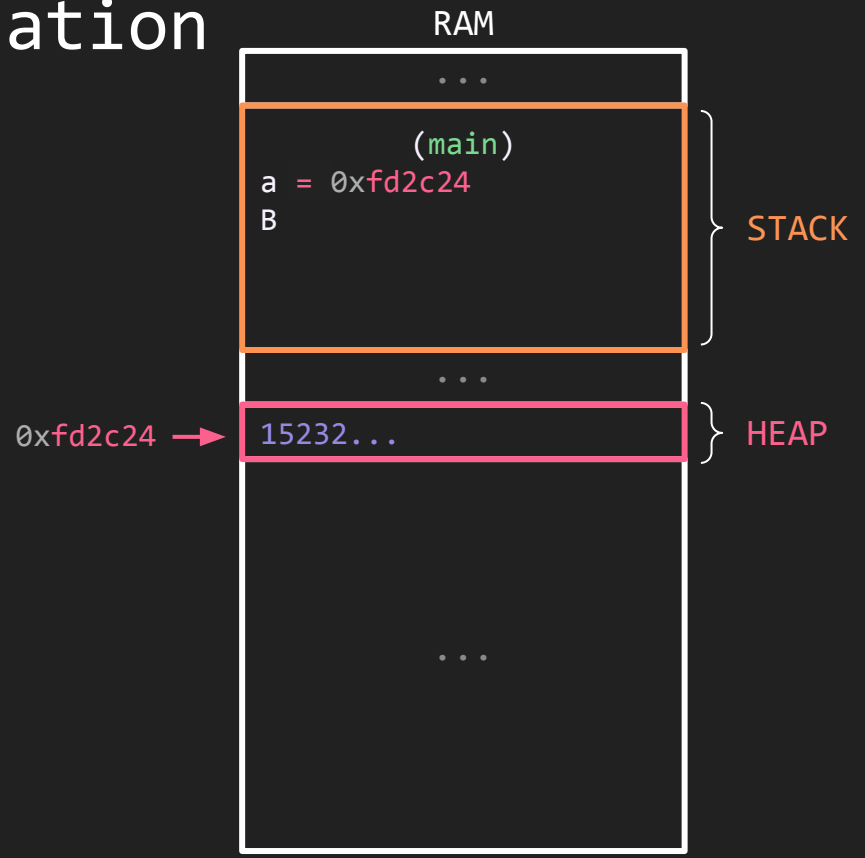
free - Memory Deallocation

```c
int* a = malloc(sizeof(int));
char *B = calloc(4, sizeof(int));
// Código
free(a);
free(B);
```



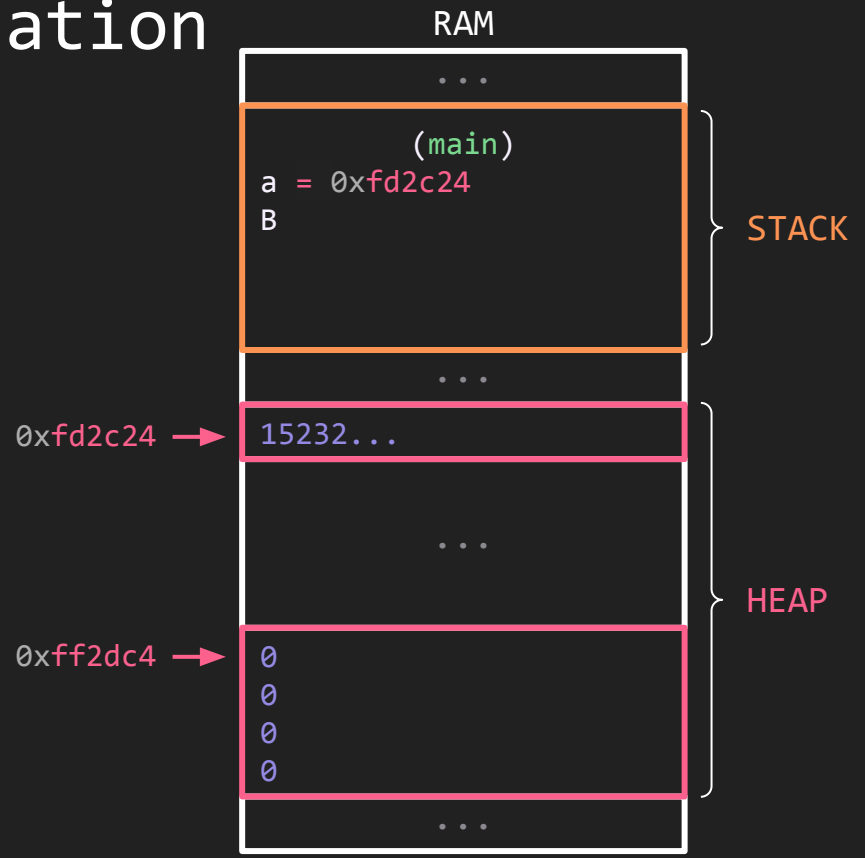
free - Memory Deallocation

```c
int* a = malloc(sizeof(int));
char *B = calloc(4, sizeof(int));
// Código
free(a);
free(B);
```



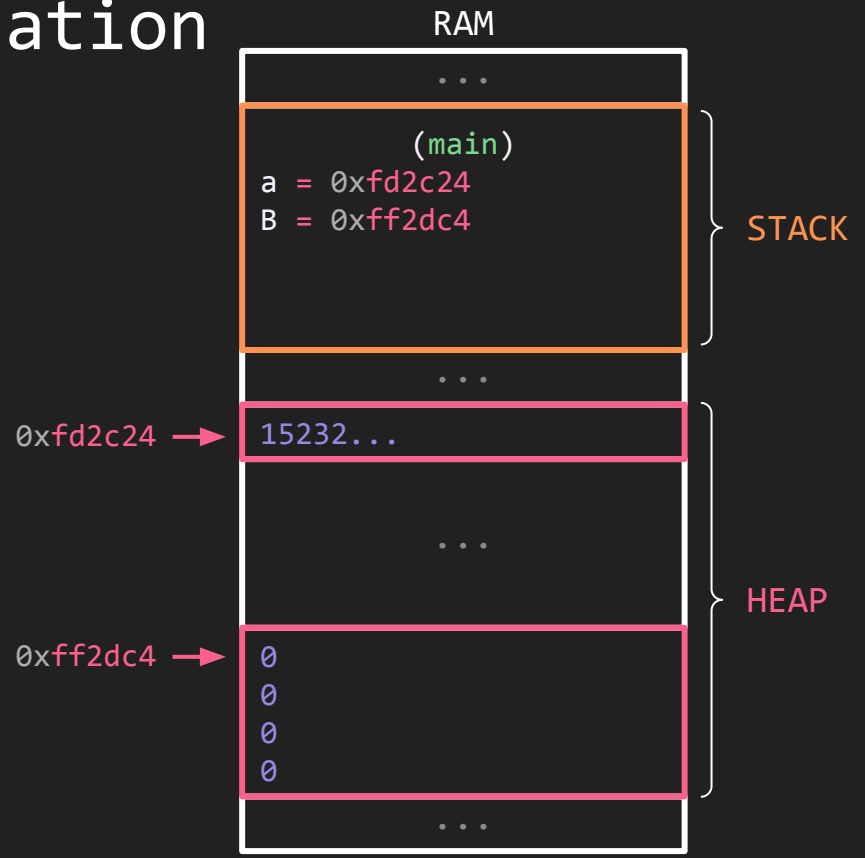
free - Memory Deallocation

```c
int* a = malloc(sizeof(int));
char *B = calloc(4, sizeof(int));
// Código
free(a);
free(B);
```



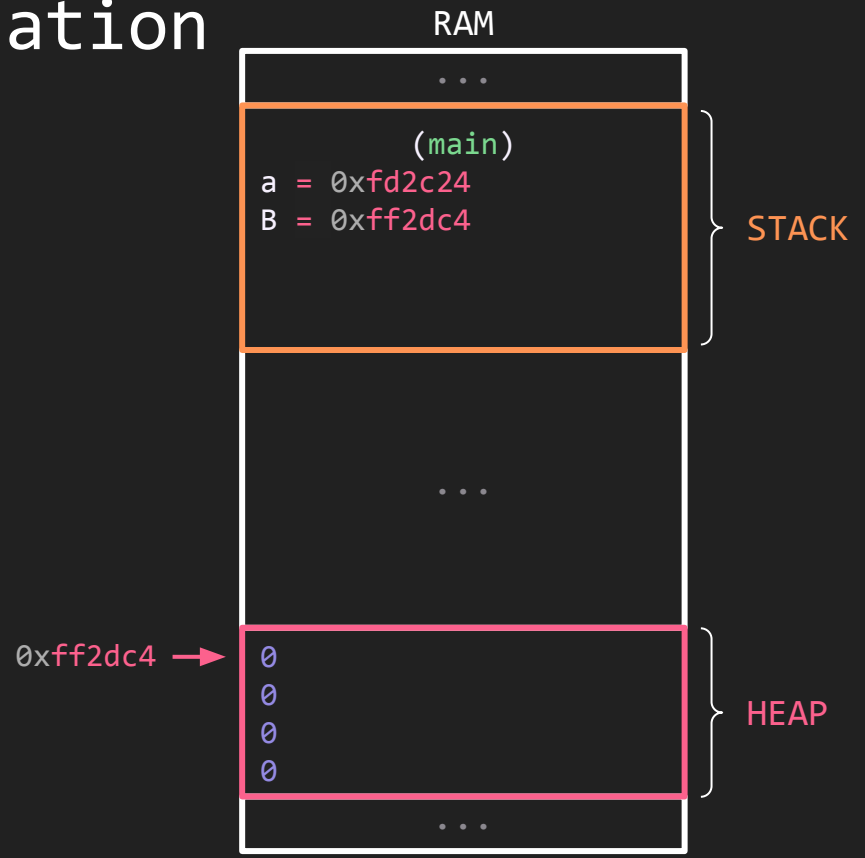
free - Memory Deallocation

```c
int* a = malloc(sizeof(int));
char *B = calloc(4, sizeof(int));
// Código
free(a);
free(B);
```



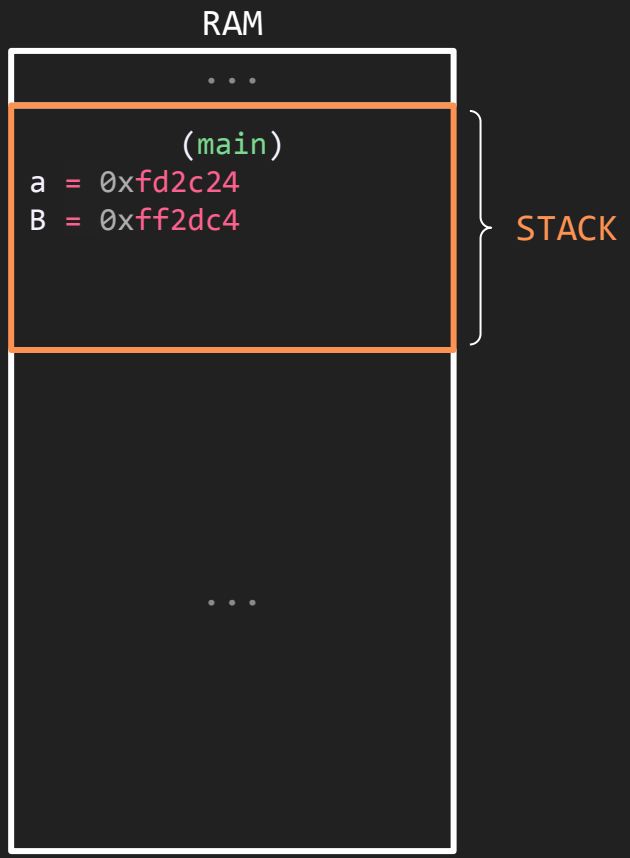
free - Memory Deallocation

```c
int* a = malloc(sizeof(int));
char *B = calloc(4, sizeof(int));
// Código
free(a);
free(B);
```



free - Memory Deallocation

```c
int* a = malloc(sizeof(int));
char *B = calloc(4, sizeof(int));
// Código
free(a);
free(B);
```



Arreglos de Arreglos en el HEAP

Para lograr esto debemos seguir los siguientes pasos:

1. Declarar *type***
2. Usar malloc/calloc con sizeof(*type**)
3. Por cada índice usar malloc/calloc con sizeof(*type*)
4. Inicializar los valores de ser necesario.

Arreglos de Arreglos

```c
int** A = calloc(2, sizeof(int*));
A[0] = calloc(3, sizeof(int));
A[1] = calloc(3, sizeof(int));
A[0][0] = 2;
A[0][1] = 27;
A[1][0] = 6;
A[1][1] = 3;
A[1][2] = 5;
```

Declarar *type***

Arreglos de Arreglos

```c
int** A = calloc(2, sizeof(int*));
A[0] = calloc(3, sizeof(int));
A[1] = calloc(3, sizeof(int));
A[0][0] = 2;
A[0][1] = 27;
A[1][0] = 6;
A[1][1] = 3;
A[1][2] = 5;
```

Usar malloc/calloc con sizeof(*type**).

Arreglos de Arreglos

```c
int** A = calloc(2, sizeof(int*));
A[0] = calloc(3, sizeof(int));
A[1] = calloc(3, sizeof(int));
A[0][0] = 2;
A[0][1] = 27;
A[1][0] = 6;
A[1][1] = 3;
A[1][2] = 5;
```

Por cada índice usar `malloc`/`calloc` con `sizeof(type)`.

Arreglos de Arreglos

```c
int** A = calloc(2, sizeof(int*));
A[0] = calloc(3, sizeof(int));
A[1] = calloc(3, sizeof(int));
A[0][0] = 2;
A[0][1] = 27;
A[1][0] = 6;
A[1][1] = 3;
A[1][2] = 5;
```

Inicializar los valores de ser necesario.

Arreglos de Arreglos

```c
int** A = calloc(2, sizeof(int*));
A[0] = calloc(3, sizeof(int));
A[1] = calloc(3, sizeof(int));
A[0][0] = 2;
A[0][1] = 27;
A[1][0] = 6;
A[1][1] = 3;
A[1][2] = 5;
```

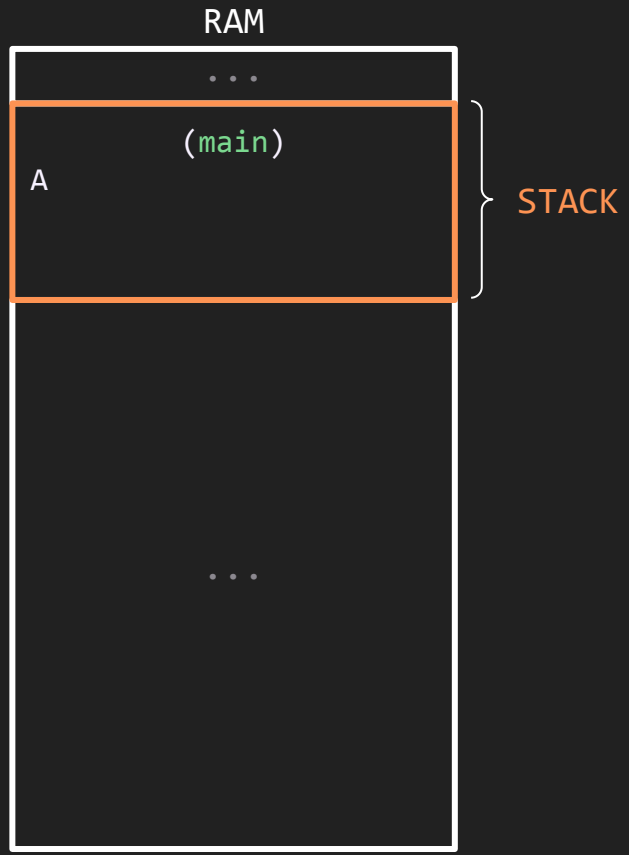
*int*** A es un puntero de punteros.

A[0] y A[1] son punteros de *int*.

Gracias a la aritmética de punteros, podemos utilizar a A al igual que a un arreglo.

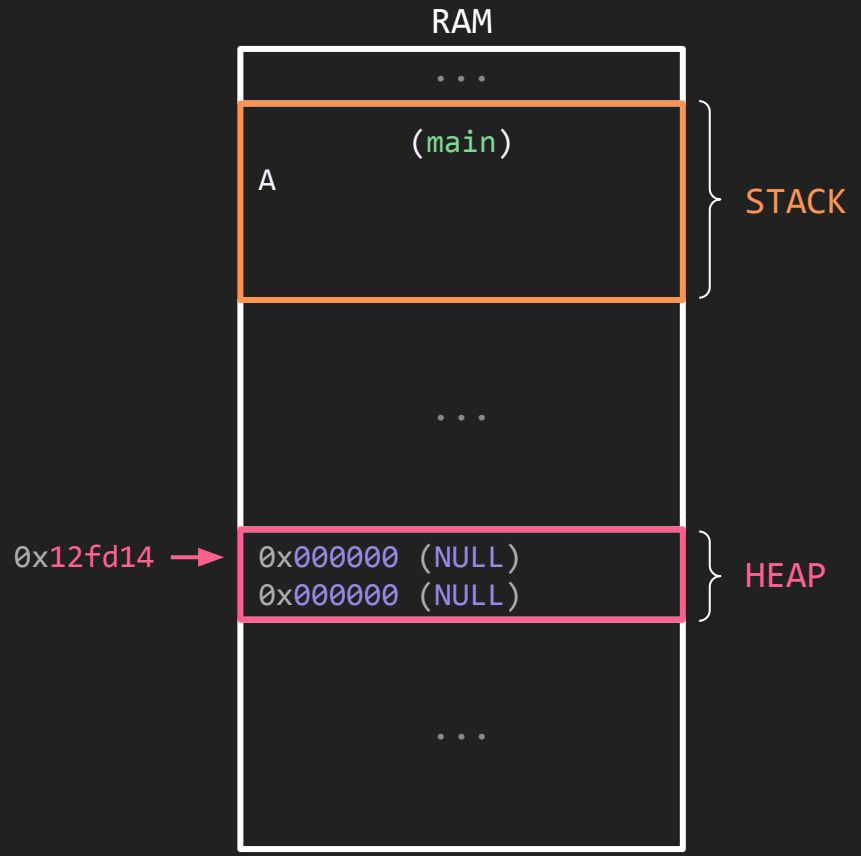
Arreglos de Arreglos

```c
int** A = calloc(2, sizeof(int*));
A[0] = calloc(3, sizeof(int));
A[1] = calloc(3, sizeof(int));
A[0][0] = 2;
A[0][1] = 27;
A[1][0] = 6;
A[1][1] = 3;
A[1][2] = 5;
```



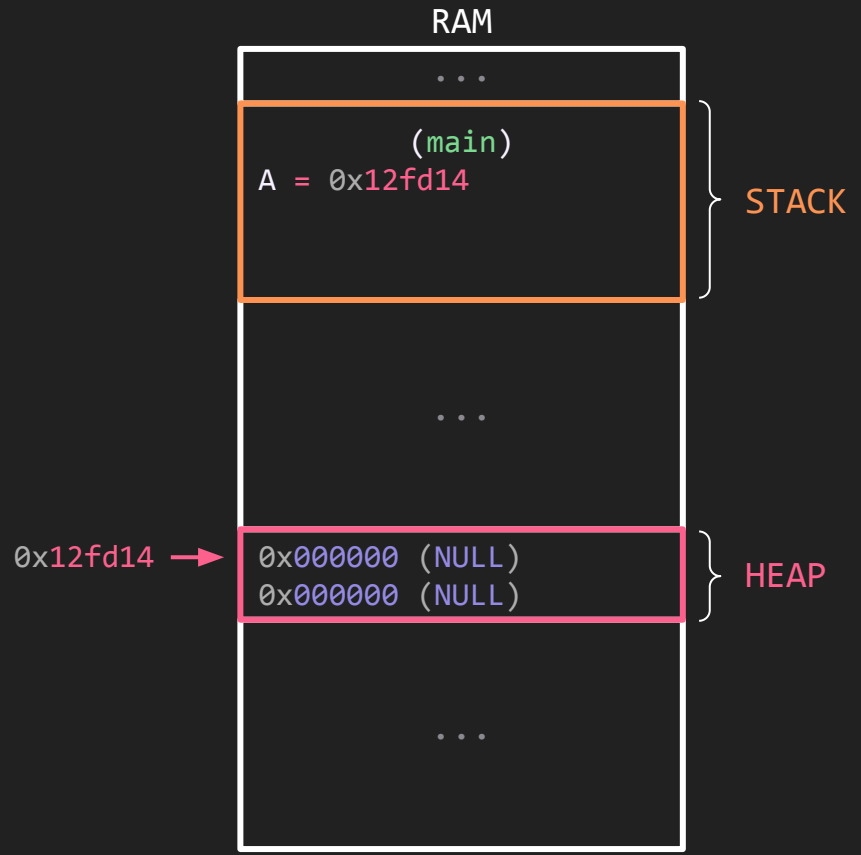
Arreglos de Arreglos

```c
int** A = calloc(2, sizeof(int*));
A[0] = calloc(3, sizeof(int));
A[1] = calloc(3, sizeof(int));
A[0][0] = 2;
A[0][1] = 27;
A[1][0] = 6;
A[1][1] = 3;
A[1][2] = 5;
```



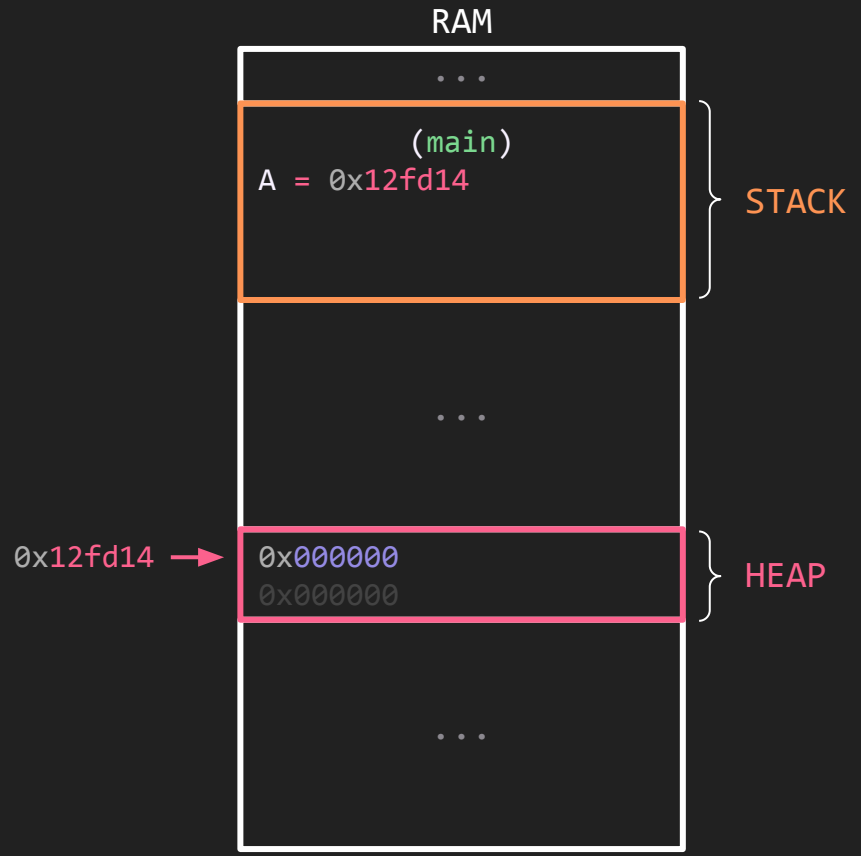
Arreglos de Arreglos

```c
int** A = calloc(2, sizeof(int*));
A[0] = calloc(3, sizeof(int));
A[1] = calloc(3, sizeof(int));
A[0][0] = 2;
A[0][1] = 27;
A[1][0] = 6;
A[1][1] = 3;
A[1][2] = 5;
```



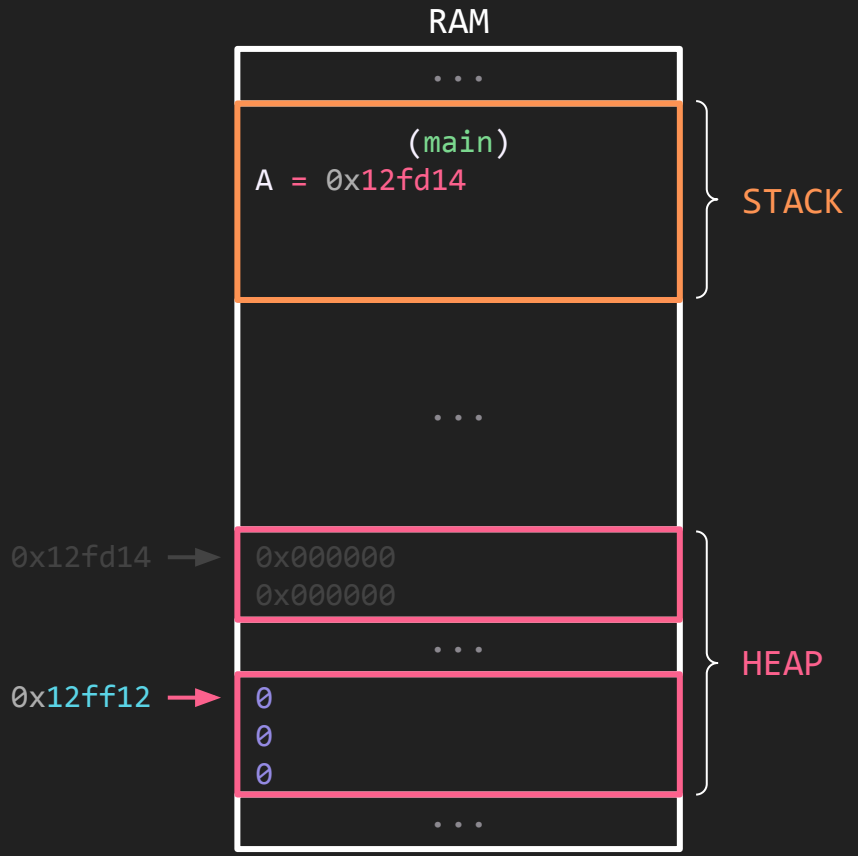
Arreglos de Arreglos

```c
int** A = calloc(2, sizeof(int*));
A[0] = calloc(3, sizeof(int));
A[1] = calloc(3, sizeof(int));
A[0][0] = 2;
A[0][1] = 27;
A[1][0] = 6;
A[1][1] = 3;
A[1][2] = 5;
```



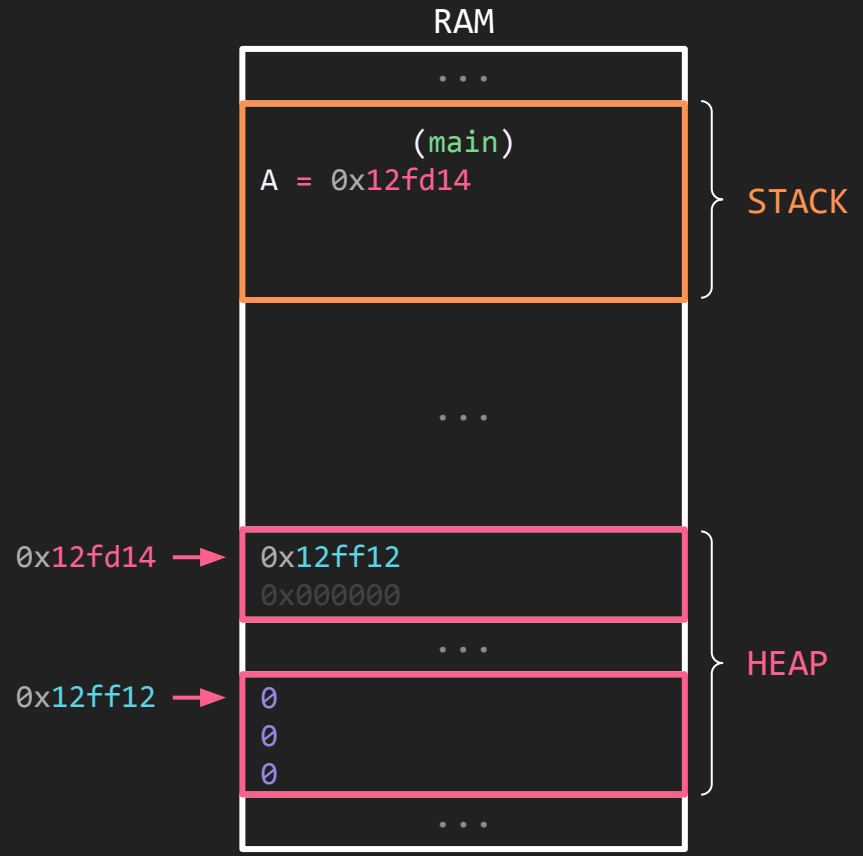
Arreglos de Arreglos

```c
int** A = calloc(2, sizeof(int*));
A[0] = calloc(3, sizeof(int));
A[1] = calloc(3, sizeof(int));
A[0][0] = 2;
A[0][1] = 27;
A[1][0] = 6;
A[1][1] = 3;
A[1][2] = 5;
```



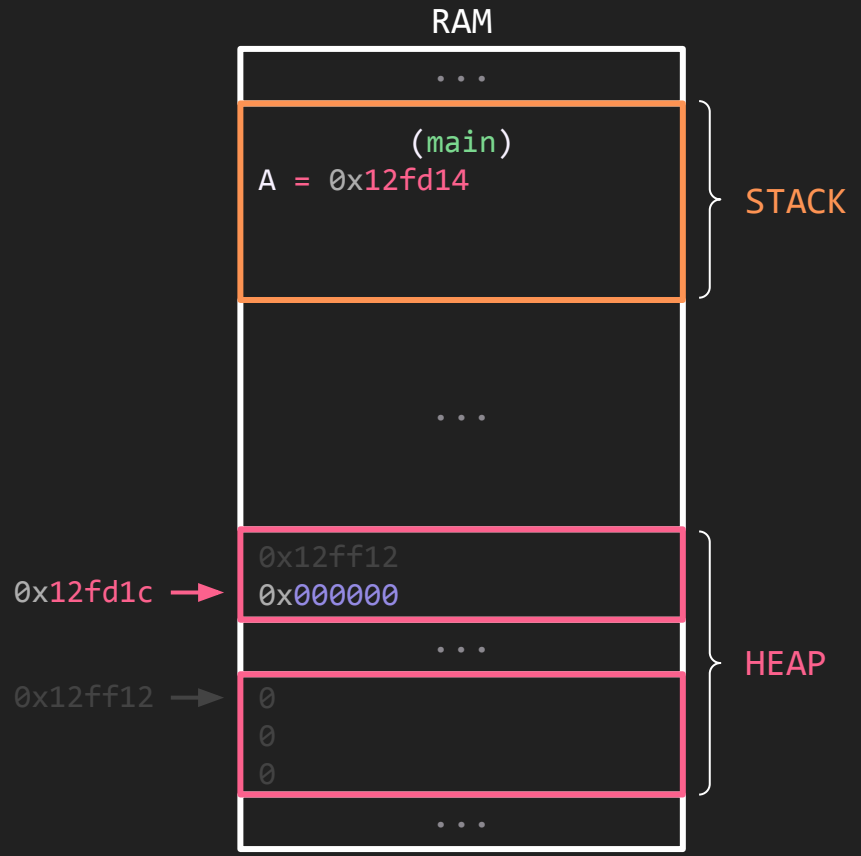
Arreglos de Arreglos

```c
int** A = calloc(2, sizeof(int*));
A[0] = calloc(3, sizeof(int));
A[1] = calloc(3, sizeof(int));
A[0][0] = 2;
A[0][1] = 27;
A[1][0] = 6;
A[1][1] = 3;
A[1][2] = 5;
```



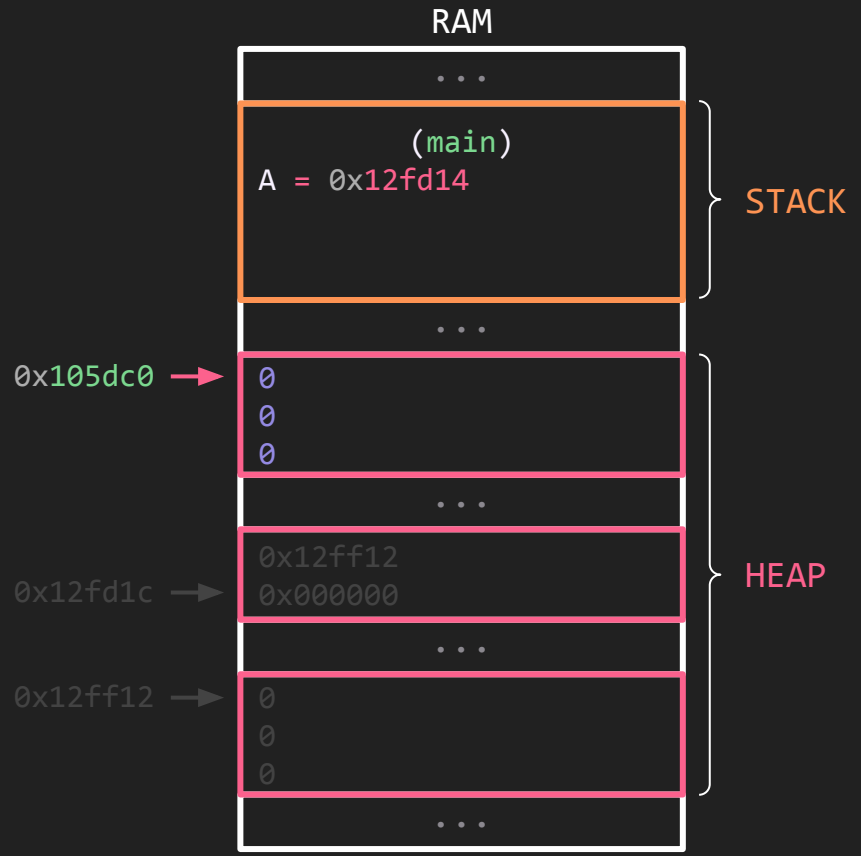
Arreglos de Arreglos

```c
int** A = calloc(2, sizeof(int*));
A[0] = calloc(3, sizeof(int));
A[1] = calloc(3, sizeof(int));
A[0][0] = 2;
A[0][1] = 27;
A[1][0] = 6;
A[1][1] = 3;
A[1][2] = 5;
```



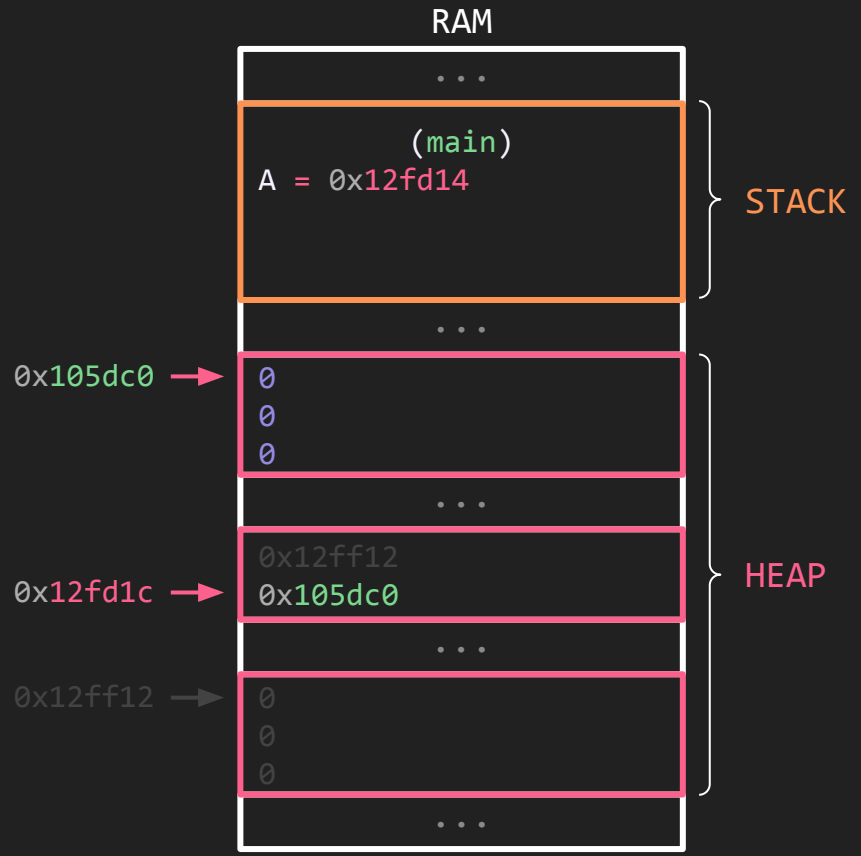
Arreglos de Arreglos

```
int** A = calloc(2, sizeof(int*));
A[0] = calloc(3, sizeof(int));
A[1] = calloc(3, sizeof(int));
A[0][0] = 2;
A[0][1] = 27;
A[1][0] = 6;
A[1][1] = 3;
A[1][2] = 5;
```



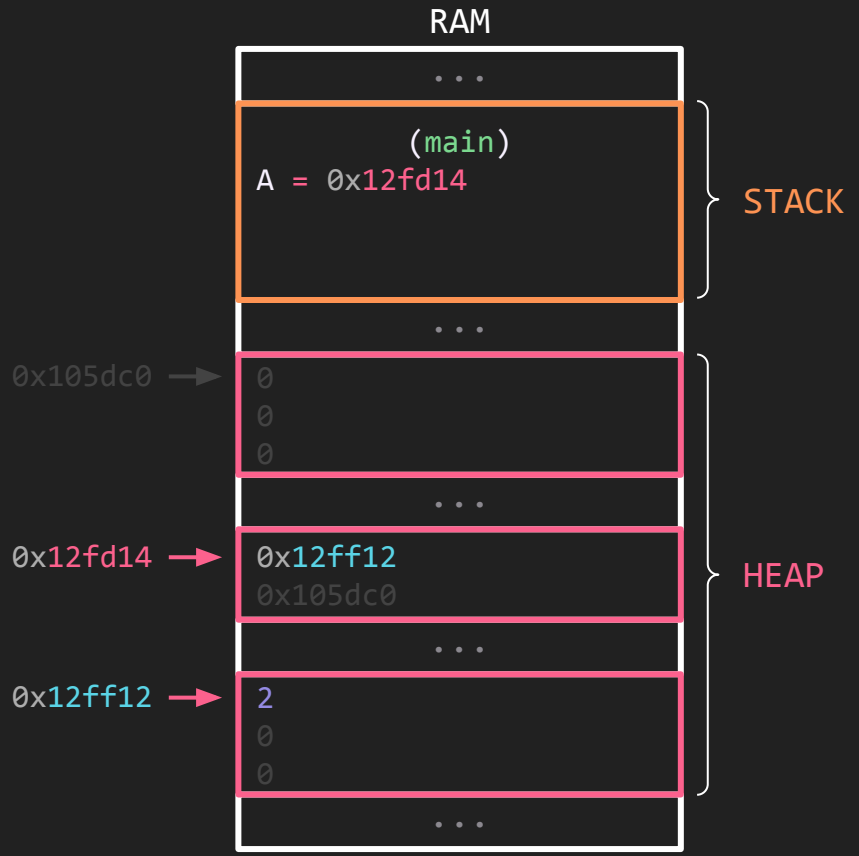
Arreglos de Arreglos

```
int** A = calloc(2, sizeof(int*));
A[0] = calloc(3, sizeof(int));
A[1] = calloc(3, sizeof(int));
A[0][0] = 2;
A[0][1] = 27;
A[1][0] = 6;
A[1][1] = 3;
A[1][2] = 5;
```



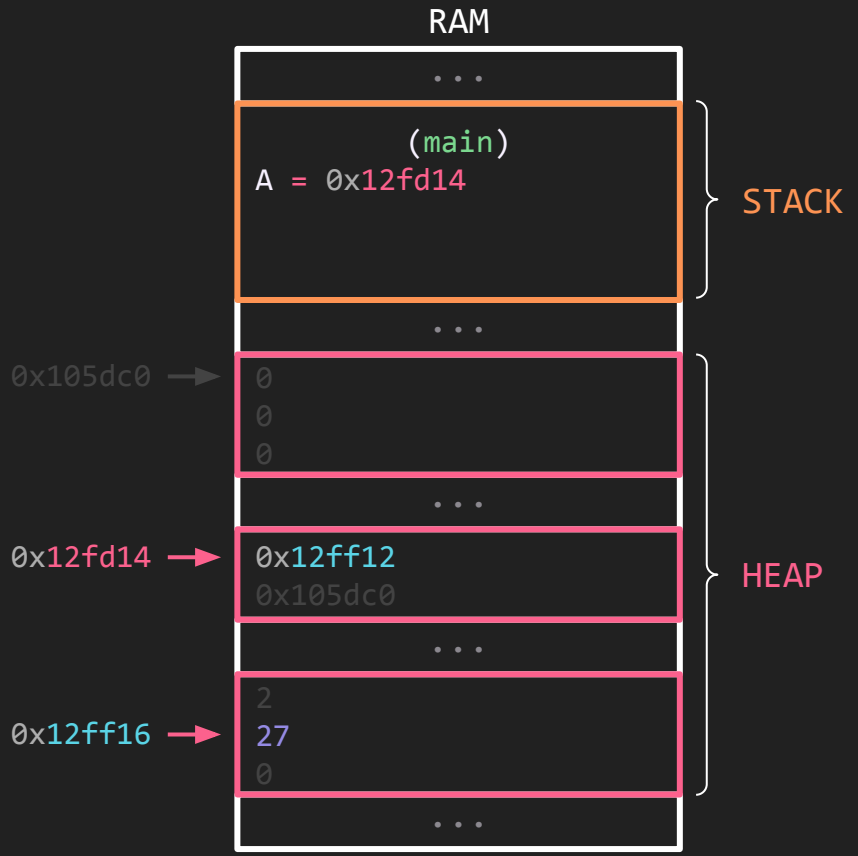
Arreglos de Arreglos

```c
int** A = calloc(2, sizeof(int*));
A[0] = calloc(3, sizeof(int));
A[1] = calloc(3, sizeof(int));
A[0][0] = 2;
A[0][1] = 27;
A[1][0] = 6;
A[1][1] = 3;
A[1][2] = 5;
```



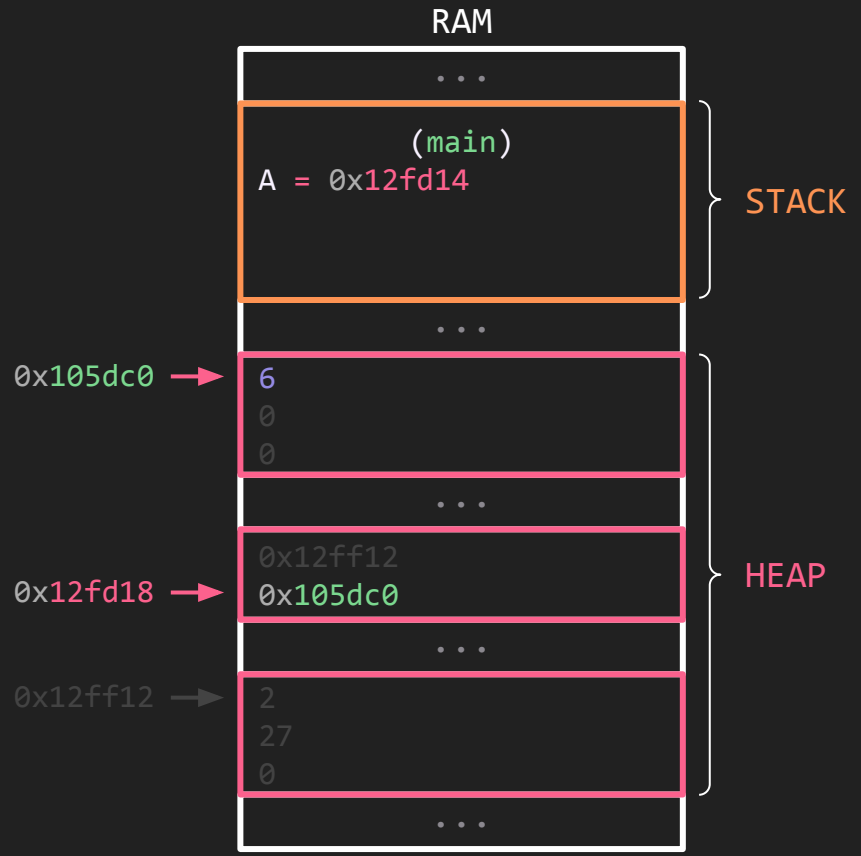
Arreglos de Arreglos

```c
int** A = calloc(2, sizeof(int*));
A[0] = calloc(3, sizeof(int));
A[1] = calloc(3, sizeof(int));
A[0][0] = 2;
A[0][1] = 27;
A[1][0] = 6;
A[1][1] = 3;
A[1][2] = 5;
```



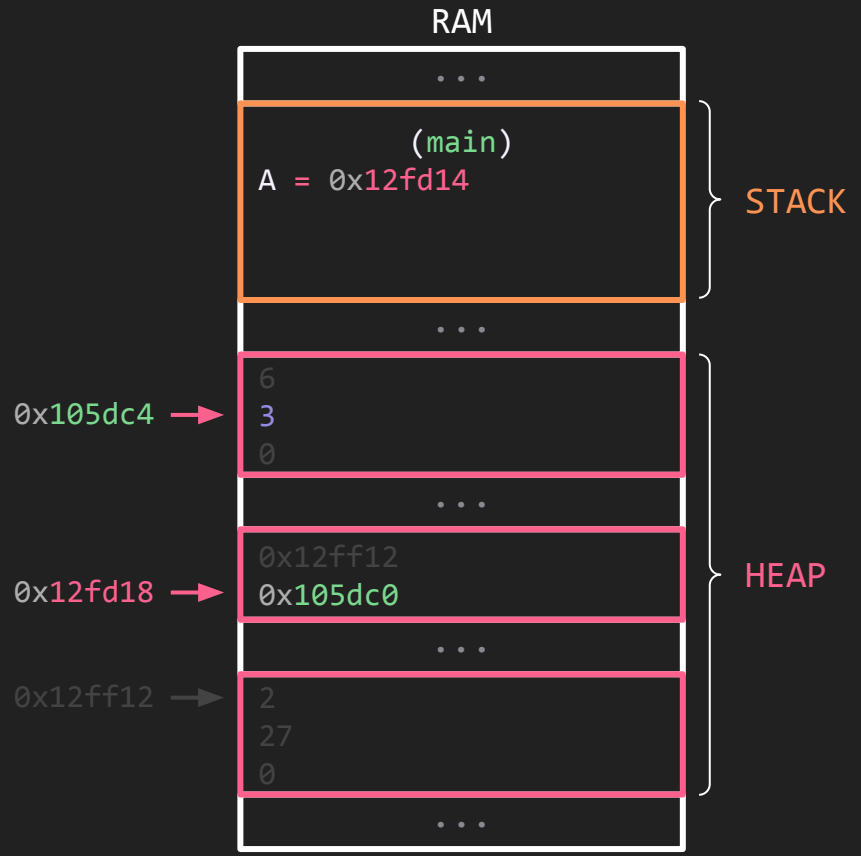
Arreglos de Arreglos

```c
int** A = calloc(2, sizeof(int*));
A[0] = calloc(3, sizeof(int));
A[1] = calloc(3, sizeof(int));
A[0][0] = 2;
A[0][1] = 27;
A[1][0] = 6;
A[1][1] = 3;
A[1][2] = 5;
```



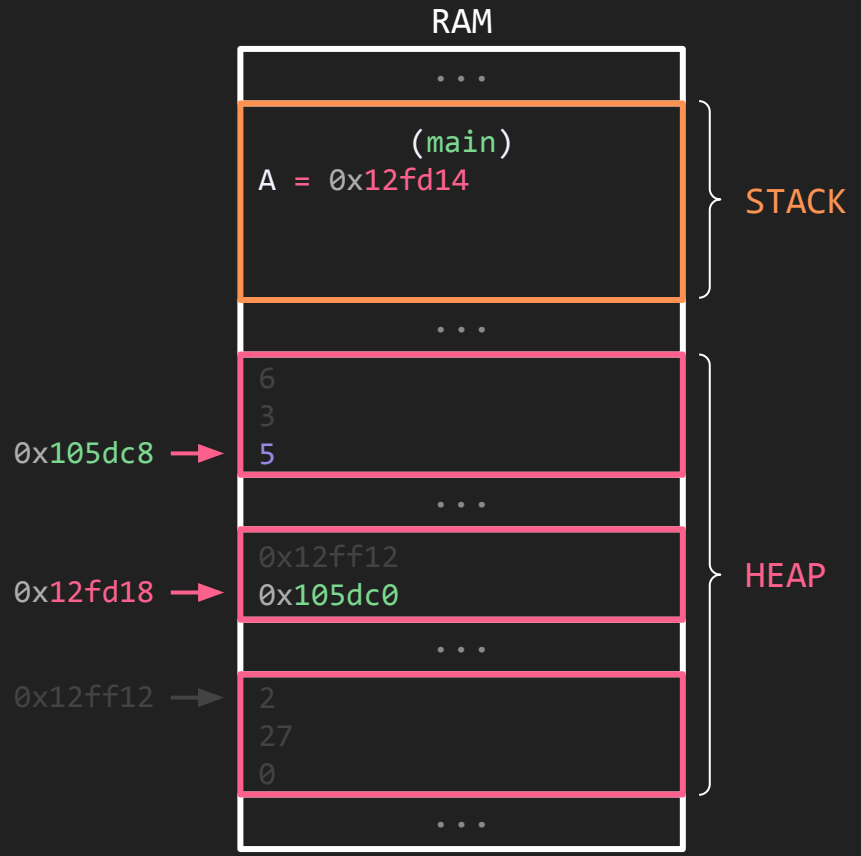
Arreglos de Arreglos

```c
int** A = calloc(2, sizeof(int*));
A[0] = calloc(3, sizeof(int));
A[1] = calloc(3, sizeof(int));
A[0][0] = 2;
A[0][1] = 27;
A[1][0] = 6;
A[1][1] = 3;
A[1][2] = 5;
```



Arreglos de Arreglos

```c
int** A = calloc(2, sizeof(int*));
A[0] = calloc(3, sizeof(int));
A[1] = calloc(3, sizeof(int));
A[0][0] = 2;
A[0][1] = 27;
A[1][0] = 6;
A[1][1] = 3;
A[1][2] = 5;
```



Errores de Memoria

Errores de memoria

Suceden debido a un incorrecto manejo de memoria, ya sea en el HEAP o en el STACK.

Acceso inválido

```c
int* a = malloc(sizeof(int));
*a = 5;
free(a);
int b = 2 + *a;
```

```c
int* a = malloc(2 * sizeof(int));
a[0] = 3;
a[1] = 2;
int b = 2 + a[3];
```

Este error sucede cuando se intenta acceder a un puntero que fue liberado, o a un sector de memoria erróneo.

El programa no se cae siempre, pero da resultados erróneos.

Doble liberación

```c
int* a = malloc(sizeof(int));
*a = 5;
free(a);
free(a);
```

Este error sucede cuando se libera un puntero que ya había sido liberado previamente.

El programa no se cae siempre, pero hay ataques maliciosos que se aprovechan de este error.

Memoria no inicializada

```c
int* a = malloc(sizeof(int));
int b = 2 + *a;
```

Este error sucede cuando se intenta acceder a una variable o puntero que no ha sido inicializado.

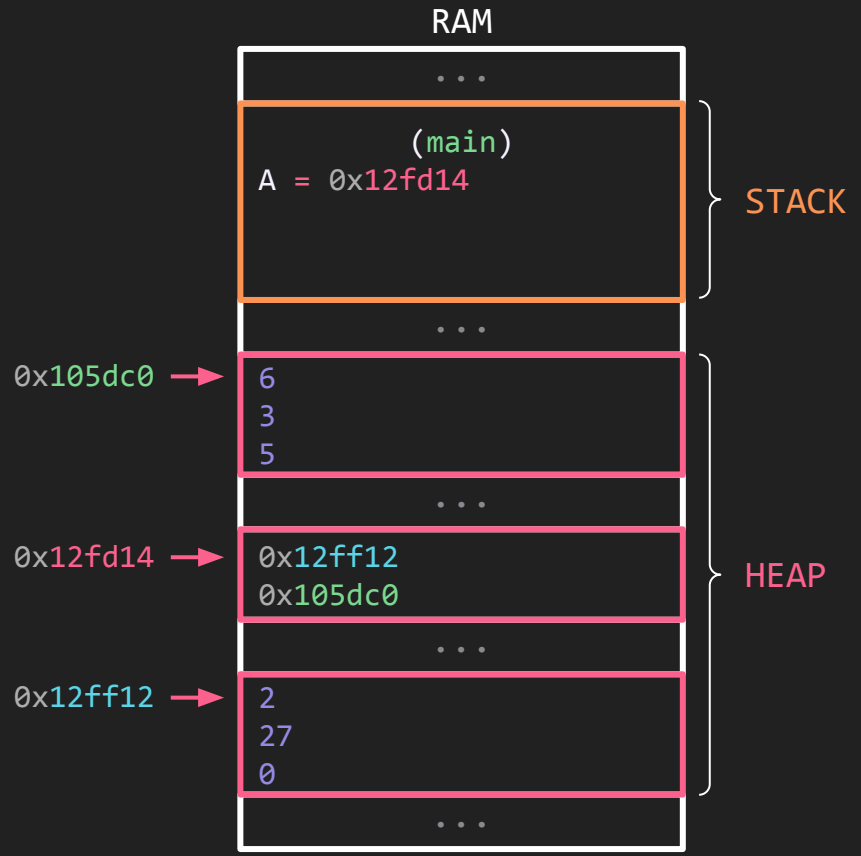
El programa no se cae siempre, pero da resultados erróneos.

Memory Leaks

Este error sucede cuando no se libera la memoria del HEAP.

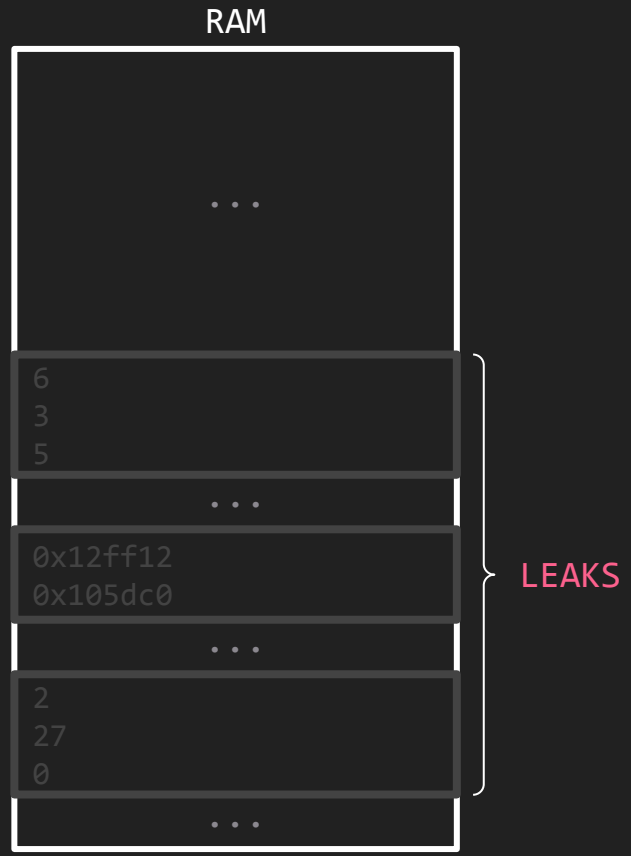
Memory Leaks

```c
int** A = calloc(2, sizeof(int*));
A[0] = calloc(3, sizeof(int));
A[1] = calloc(3, sizeof(int));
A[0][0] = 2;
A[0][1] = 27;
A[1][0] = 6;
A[1][1] = 3;
A[1][2] = 5;
// faltan los free
```



Memory Leaks

La memoria que no fue liberada se pierde y el sistema operativo se puede demorar mucho en recuperarla.



Límite del HEAP

Límite del HEAP

El HEAP no tiene límite.

Límite del HEAP

```c
int* a = malloc(1000000000);
if (!a)
{
  printf("No blocks of that size left.\n");
}
```

```
$ gcc main.c -o main
$ ./main
No blocks of that size left.
```

Límite del HEAP

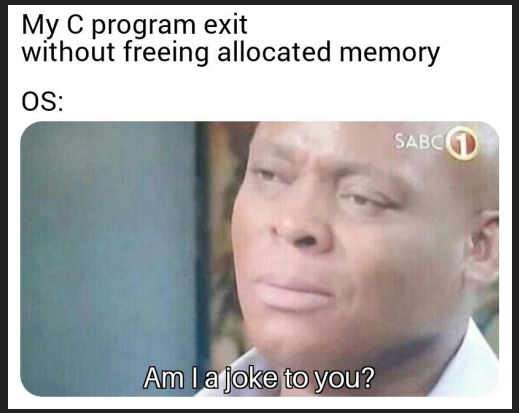
```c
int* a = malloc(1000000000);
if (!a)
{
  printf("No blocks of that size left.\n");
}
```

```
$ gcc main.c -o main
$ ./main
No blocks of that size left.
```

La petición de memoria puede fallar por falta de espacio o un bloque muy grande.

Si malloc/calloc llega a fallar, retornará el puntero NULL.

¡Muchas Gracias!



With 💜 by @vichoeq & @KnowYourselves