Fundamentos da Programação

Memória dinâmica

P.PORTO
ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Conteúdo

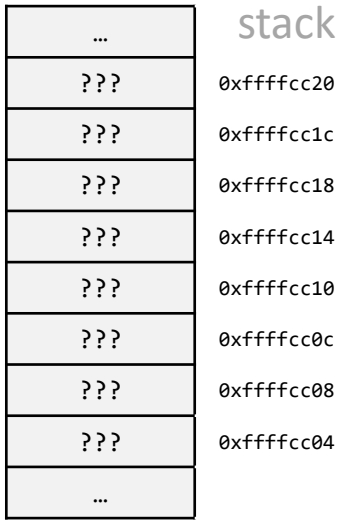
- Alocação de memória dinâmica
- Precauções
- Apontadores para apontadores
- Alocar memória em funções

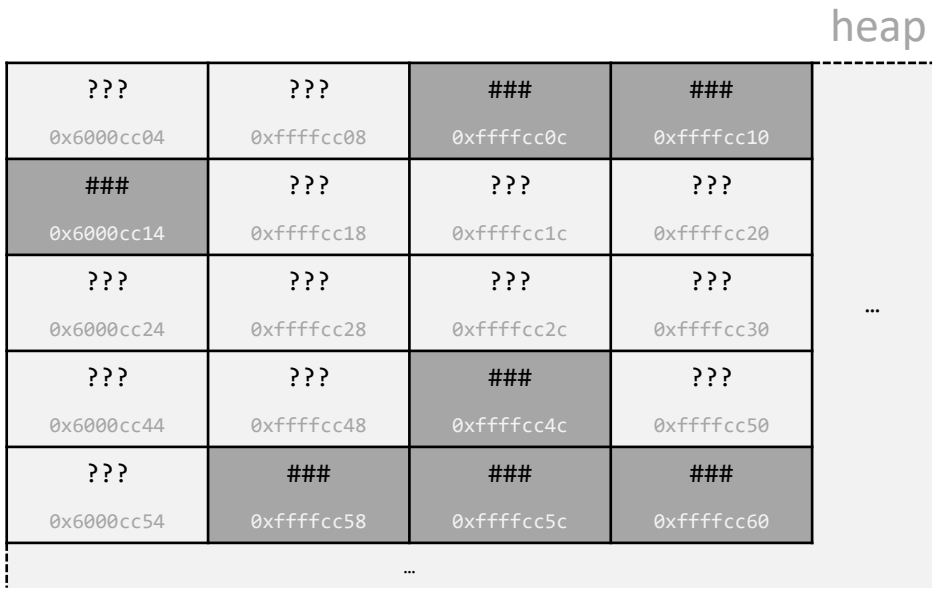
Problema

- Nos programas vistos nos capítulos anteriores, todas as necessidades de memória foram determinadas antes da execução do programa, definindo as variáveis necessárias (**alocação estática**).
  - Quando é necessário alocar um array e não sabemos, à partida, quantos elementos vão ser guardados estabelecemos um tamanho "suficientemente grande":
    - Se não chegar, o programa é limitado...
    - Se for demasiado grande, existe um desperdício de memória....
- Podem existir casos em que as necessidades de memória de um programa só podem ser determinadas durante o tempo de execução.
  - Por exemplo, quando a memória necessária depende da entrada do utilizador. Nesses casos, os programas precisam **alocar memória dinamicamente**.

Alocação dinâmica de memória

- Nestas situações podemos alocar a memória dinamicamente na **heap**





Stack vs Heap

Stack	Heap
Bloco contíguo de memória.	Memória não é necessariamente alocada sequencialmente. Pode existir fragmentação.
Limitada em termos de espaço.	Grande espaço de memória disponível para alocação de memória.
Mais rápida.	A heap é muito mais lenta que a stack.
Quando se pretende utilizar estruturas de maior dimensão, ou funções recursivas mais intrincadas, podem ocorrer erros do tipo Stack Overflow.	Memória alocada dinamicamente tem que ser explicitamente libertada. Caso contrário ocorre uma memory leak ou fuga de memória
Elementos estão empilhados, e apenas se pode inserir ou remover no topo da pilha. Quando um elemento da stack é removido, os seus recursos são automaticamente libertados. Insere-se no topo e remove-se no topo (LIFO – Last In, First Out).	Sistema operativo tem que gerir acessos à heap. Memória reservada existe durante toda a "vida" do programa, ou até que seja explicitamente removida

Alocação dinâmica de memória

- Processo de alocação de memória durante a execução de um programa.
- O header file **`stdlib.h`** fornece funções relacionadas com a alocação/libertação de memória:
  - **`void* malloc(size_t size)`**
    - Aloca um bloco de memória de **`size`** bytes.
  - **`void* calloc(size_t num, size_t size)`**
    - Aloca um bloco de memória de **`num`** * **`size`** bytes, e inicializa todos os seus valores a **`0`**.
  - **`void* realloc(void* ptr, size_t size)`**
    - Altera o tamanho do bloco para o qual **`ptr`** aponta, para **`size`** bytes.
  - **`void free(void* ptr)`**
    - Liberta o bloco de memória para o qual **`ptr`** aponta.

malloc

```
T *a = (T *) malloc(n * sizeof(T));
```

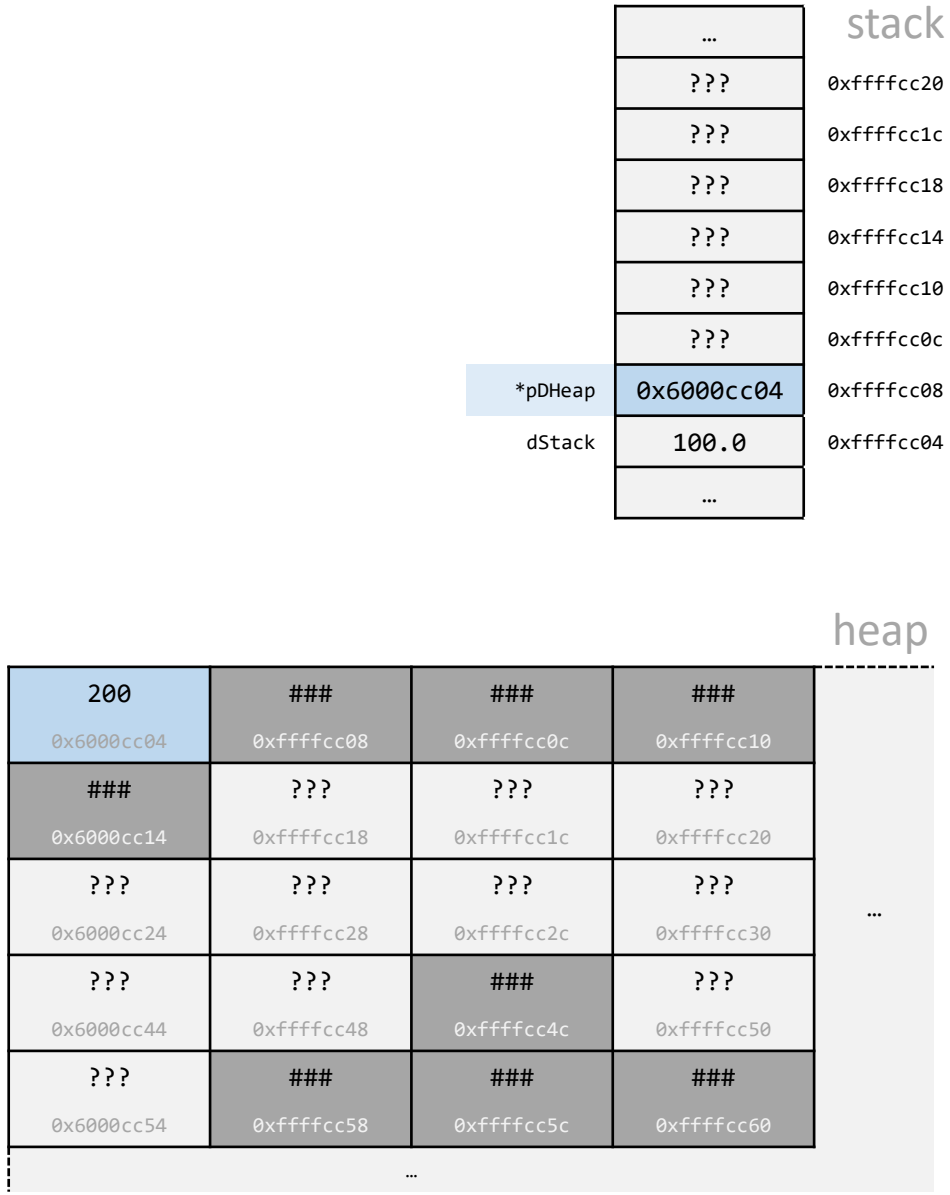
- **T** identifica o tipo de dados.
- A função **malloc** recebe um argumento que é o número de bytes de memória que pretendemos alocar.
  - Se quisermos guardar **n** inteiros temos de pedir **n** * o número de bytes que cada inteiro ocupa.
- Retorna o primeiro endereço do bloco de memória que foi reservado. Caso não seja possível fazer a alocação de memória, **malloc** devolve **NULL**.
  - É aconselhado verificar se a alocação de memória teve sucesso.
- É necessário o cast **(T *)** para o tipo apropriado, para se poder utilizar aritmética de apontadores.

malloc

Alocar um valor

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

int main() {
  double dStack = 100.0;
  double *pDHeap = (double*) malloc(sizeof (double));

  *pDHeap = 200.0;

  printf("%lf %lf", dStack, *pDHeap);

  free(pDHeap);
  pDHeap = NULL;

  return 0;
}
```

```
100.000000 200.000000
```

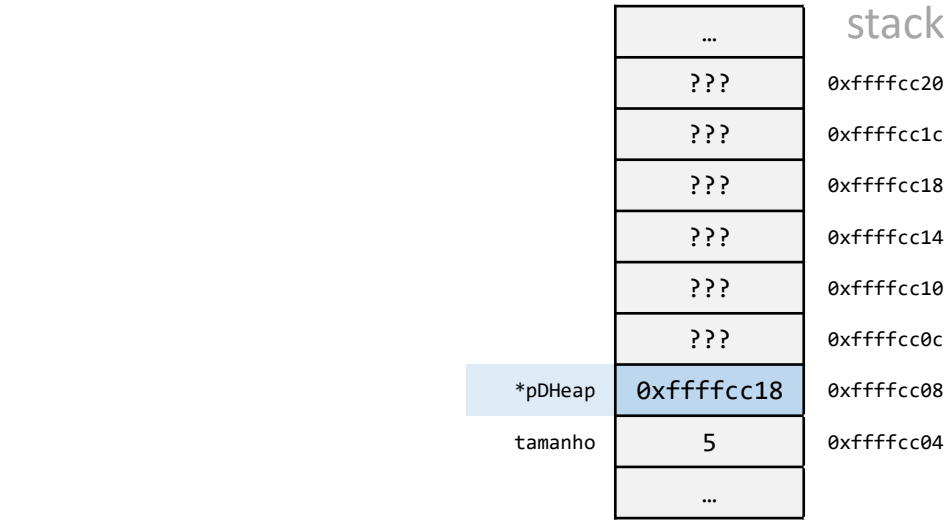


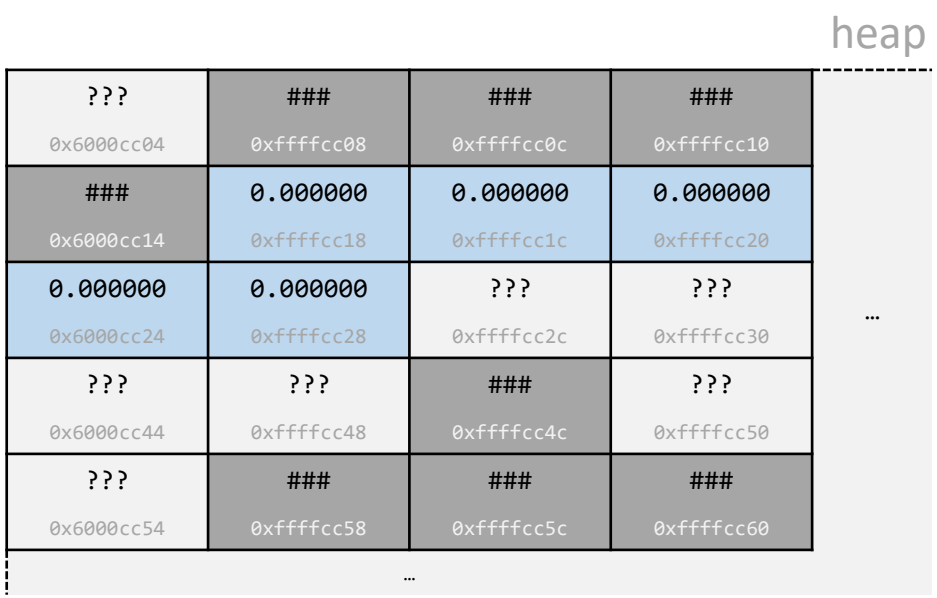
malloc

Alocar um vetor

```
…
int i, tam;
double *pDHeap;

puts("Tamanho do vetor: ");
scanf("%d", &tam);

pDHeap = (double*) malloc(sizeof(double) * tam);

for (i = 0; i < tam; i++) {
    pDHeap[i] = 0; // equiv. a *(pDHeap + i) = 0;
    printf("%lf ", pDHeap[i]);
}

free(pDHeap);
pDHeap = NULL;
…
```

```
Tamanho do vetor:
5

0.000000 0.000000 0.000000 0.000000 0.000000
```





calloc

```
T *a = (T *) calloc(n, sizeof(T));
```

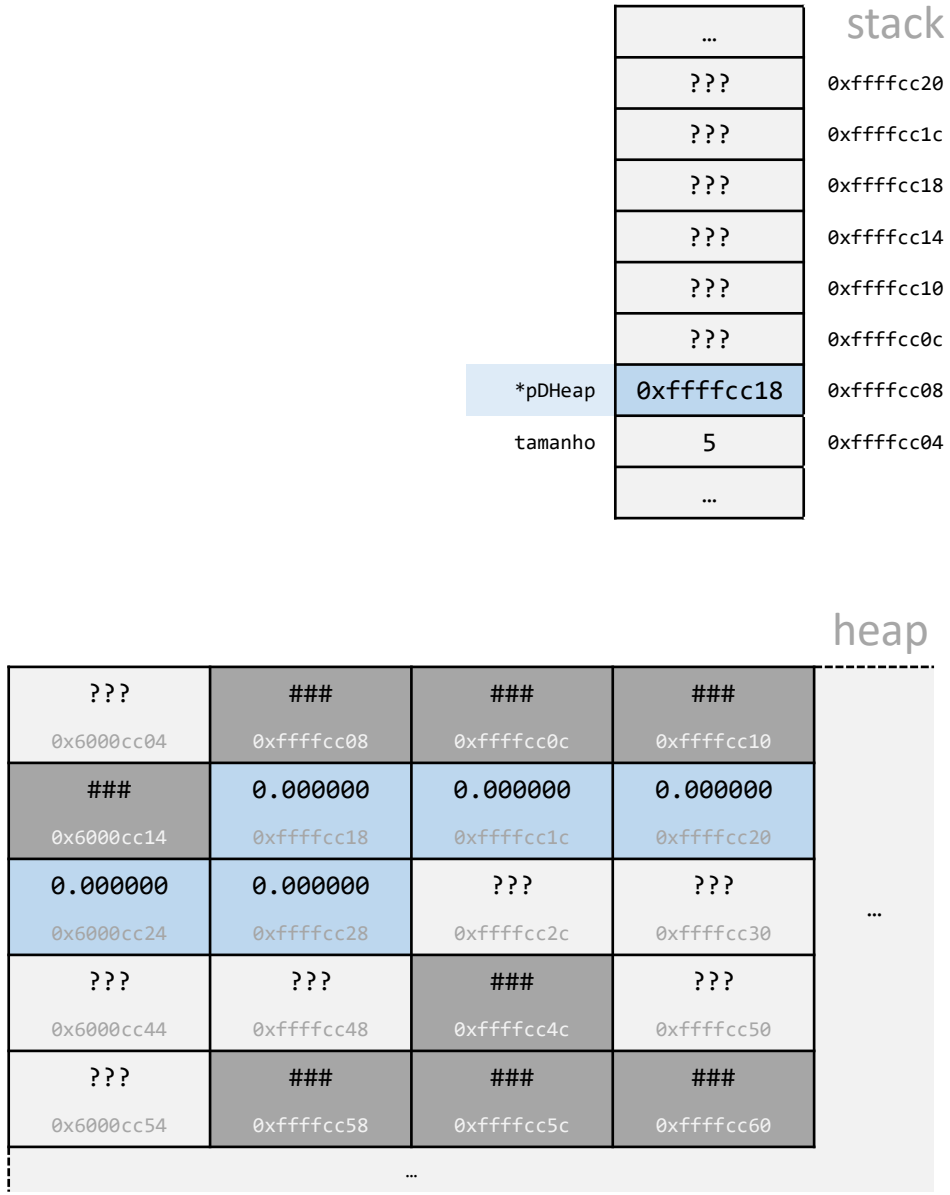
- Semelhante ao **`malloc`**, mas os blocos são inicializados a **`0`**.

calloc

Alocar um vetor

```
…
int i, tam;
double *pDHeap;

puts("Tamanho do vetor: ");
scanf("%d", &tam);

pDHeap = (double*) calloc(tam, sizeof(double));

for (i = 0; i < tam; i++) {
    printf("%lf ", pDHeap[i]);
}

free(pDHeap);
pDHeap = NULL;
…
```

```
Tamanho do vetor:
5

0.000000 0.000000 0.000000 0.000000 0.000000
```



realloc

```
ptr2 = (T *) realloc(ptr1, n);
```

- Realoca espaço previamente alocado com um novo tamanho (**n**).
  - A função recebe dois argumentos: o apontador (**ptr1**) que indica o atual bloco de memória e o novo tamanho (**n**) pretendido em bytes.
  - Permite aumentar ou diminuir tamanho.
    - Se o apontador passado como parâmetro for **NULL**, funcionamento é igual ao **malloc**.
- O **realloc** pode decidir:
  - Alocar um novo bloco de memória, copiar os dados e libertar o bloco original.
  - Expandir/contrair o bloco original "no local" sem alocar um novo.

realloc

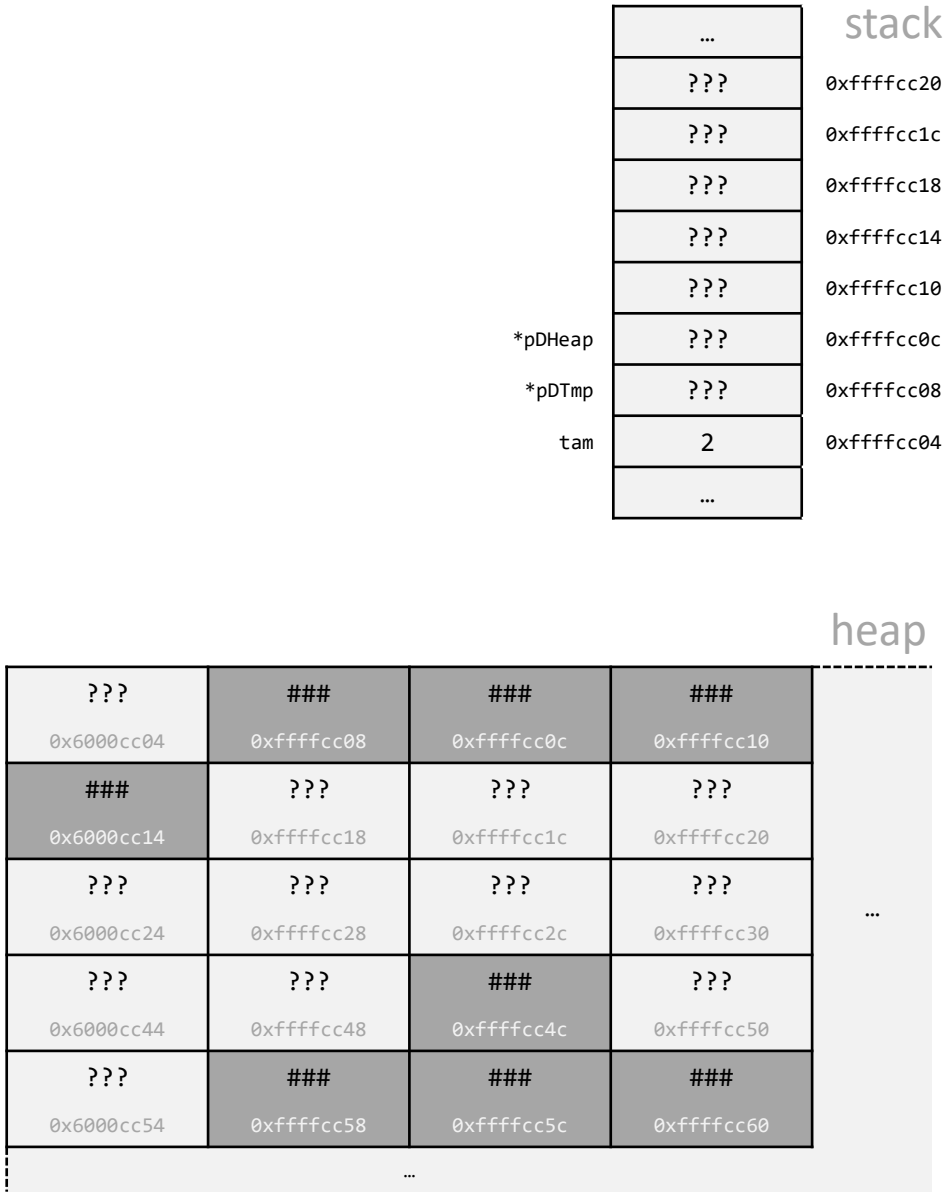
```
ptr2 = (T *) realloc(ptr1, n);
```

- O bloco realocado **contém os dados existentes** no bloco antes da alteração.
  - Caso o novo bloco seja menor, apenas são mantidos os dados existentes até ao novo tamanho definido.
- Deve verificar-se se a realocação de memória teve sucesso.
  - Se sim, a função **retorna o apontador para a primeira posição** de memória alocada.
  - Caso não seja possível fazer a realocação de memória (p.e.. falta de memória), retorna **`NULL`** e mantém o anterior bloco e apontador (**`ptr1`**) é inalterado.
    - Por esta razão, é boa prática utilizar um apontador diferente do original na realocação, e fazer a atribuição ao original apenas em caso de sucesso na realocação.

realloc

1/4

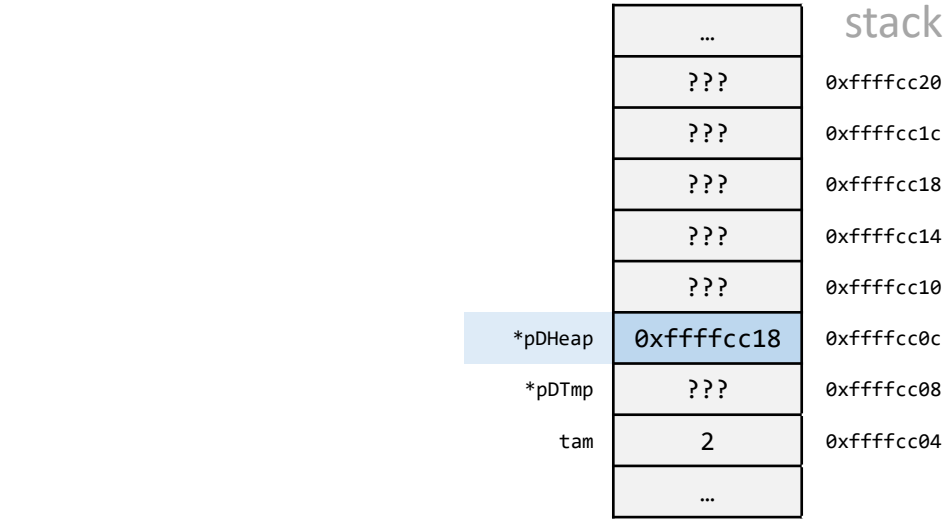
```
…
int tam = 2;
double *pDTmp, *pDHeap;

pDHeap = (double*) calloc(tam, sizeof(double));

pDTmp = (double*)
         realloc(pDHeap, sizeof(double) * (tam * 2));

if (pDTmp != NULL) {
    tam *= 2;
    pDHeap = pDTmp;
}

free(pDHeap);
pDHeap = NULL;

…
```

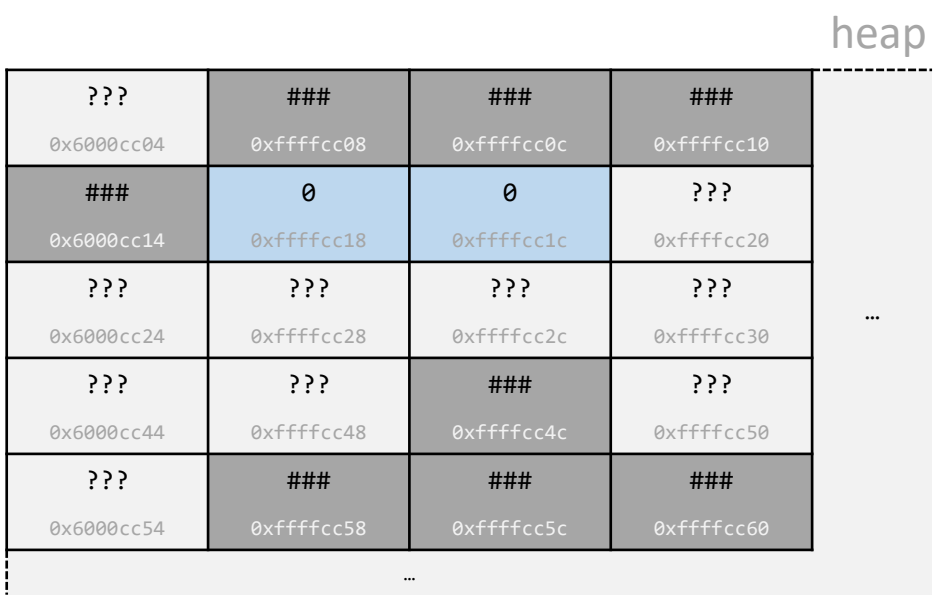


realloc

2/4

```
…
int tam = 2;
double *pDTmp, *pDHeap;

pDHeap = (double*) calloc(tam, sizeof(double));

pDTmp = (double*)
         realloc(pDHeap, sizeof(double) * (tam * 2));

if (pDTmp != NULL) {
    tam *= 2;
    pDHeap = pDTmp;
}

free(pDHeap);
pDHeap = NULL;

…
```

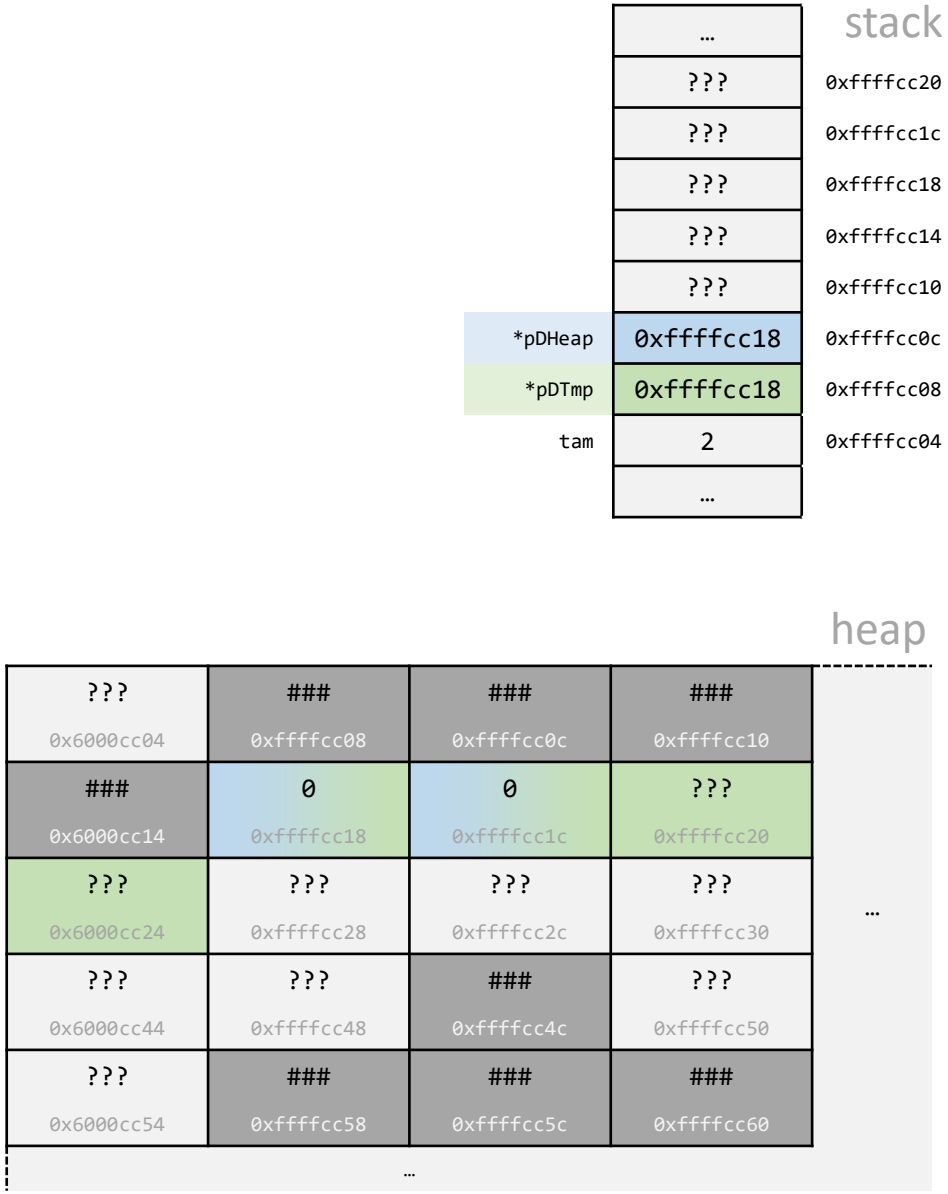




realloc

3/4

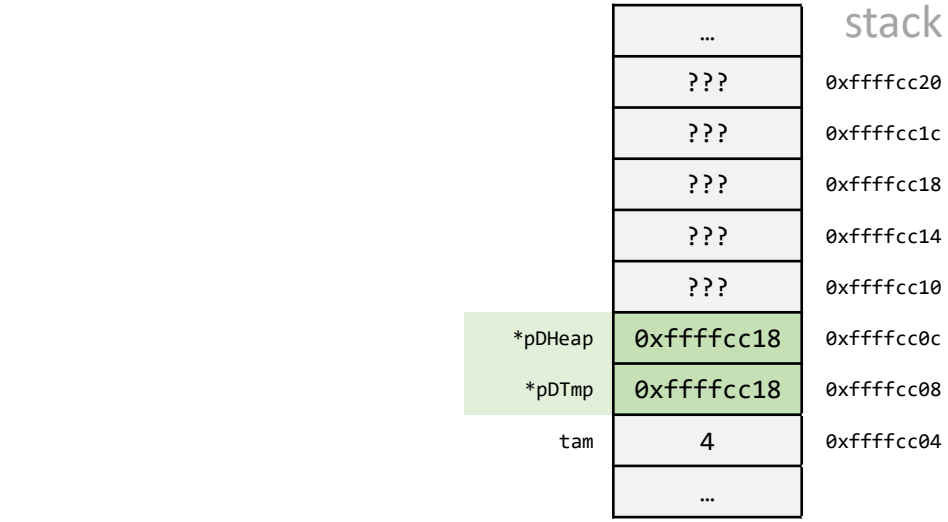
```
…
int tam = 2;
double *pDTmp, *pDHeap;

pDHeap = (double*) calloc(tam, sizeof(double));

pDTmp = (double*)
         realloc(pDHeap, sizeof(double) * (tam * 2));

if (pDTmp != NULL) {
    tam *= 2;
    pDHeap = pDTmp;
}

free(pDHeap);
pDHeap = NULL;

…
```

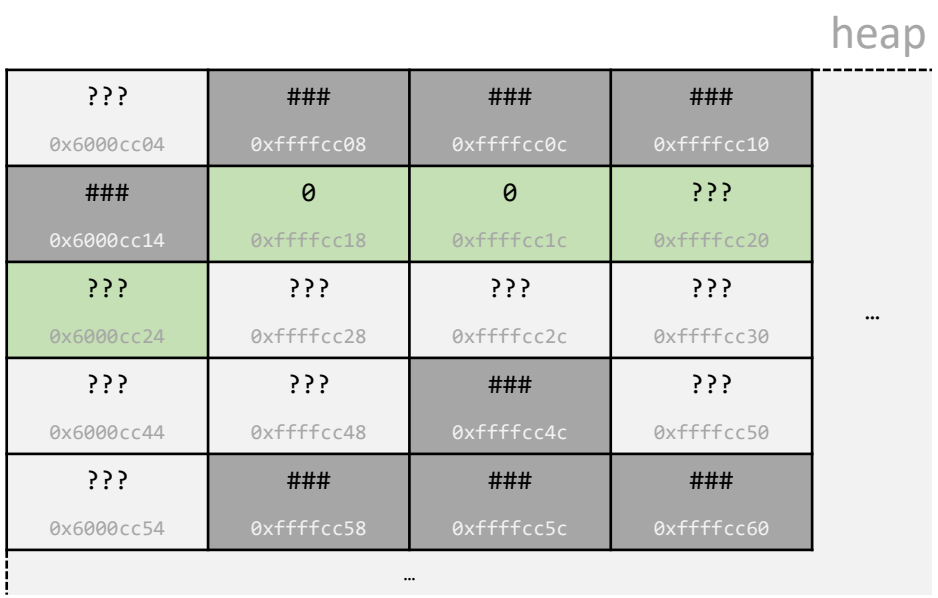


realloc

4/4

```
…
int tam = 2;
double *pDTmp, *pDHeap;

pDHeap = (double*) calloc(tam, sizeof(double));

pDTmp = (double*)
         realloc(pDHeap, sizeof(double) * (tam * 2));

if (pDTmp != NULL) {
    tam *= 2;
    pDHeap = pDTmp;
}

free(pDHeap);
pDHeap = NULL;

…
```





free

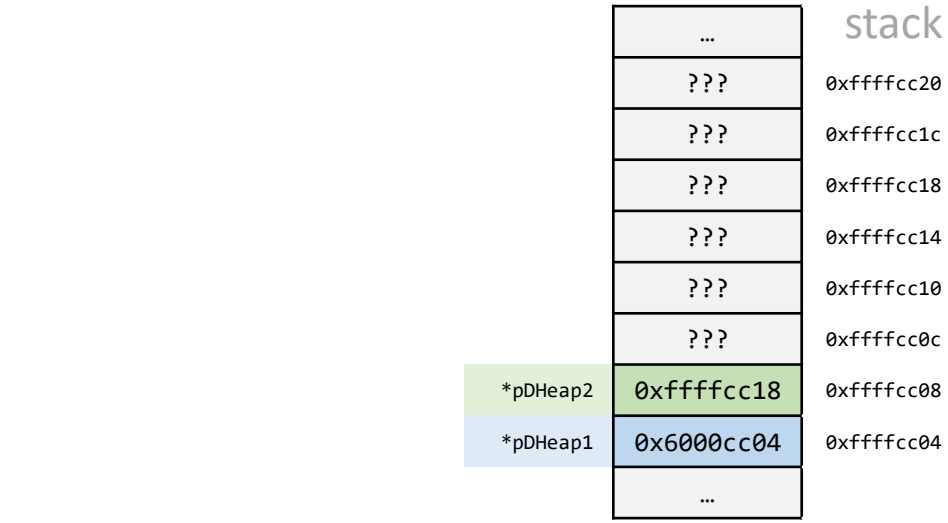
```
free(ptr)
```

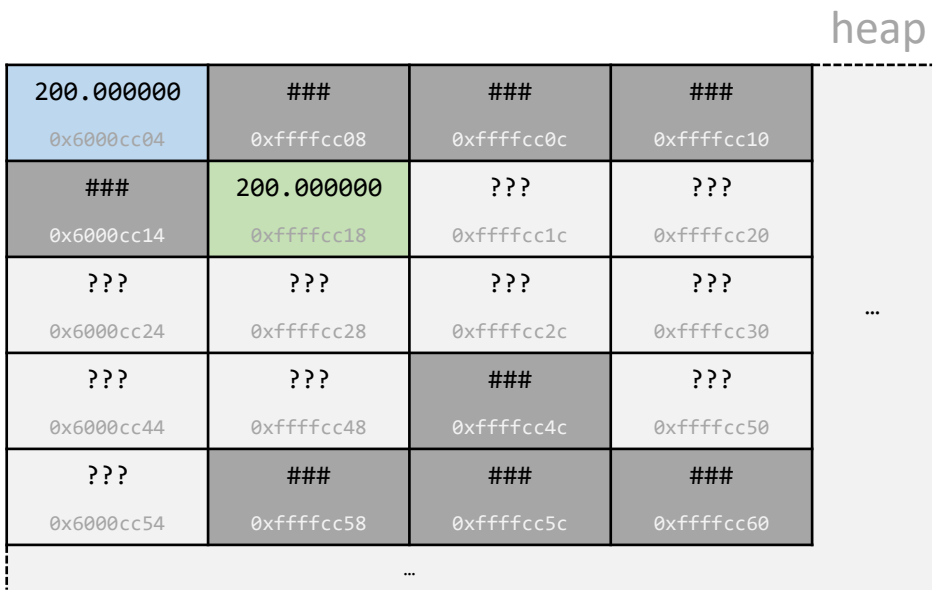
- Liberta o bloco de memória apontado por **ptr**.
- Sempre que reservamos memória dinamicamente devemos libertar essa memória quando já não for necessária.
- Para libertarmos a memória alocada dinamicamente usamos a função **free**. Uma vez libertada a memória, esta pode ser reutilizada pelo sistema operativo.
- **free** não altera o valor do apontador. Este continua a apontar para o mesmo endereço, mas esse endereço agora é inválido.
  - Para indicar que a já não aponta para nada, é boa prática atribuir **NULL** ao apontador.

free

1/3

```
…
int main() {
    double *pDHeap1 = (double*) malloc(sizeof(double));
    double *pDHeap2 = (double*) malloc(sizeof(double));

    *pDHeap1 = 200.0;
    *pDHeap2 = 200.0;

    free(pDHeap1);

    pDHeap1 = NULL;

    free(pDHeap2);

    printf("%p %p %lf", pDHeap1, pDHeap2, *pDHeap2);

    return 0;
}
```

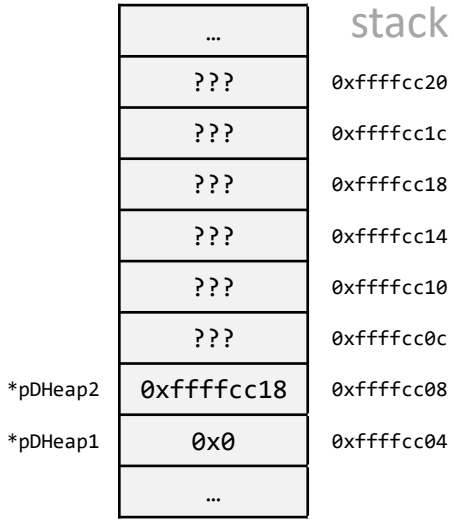


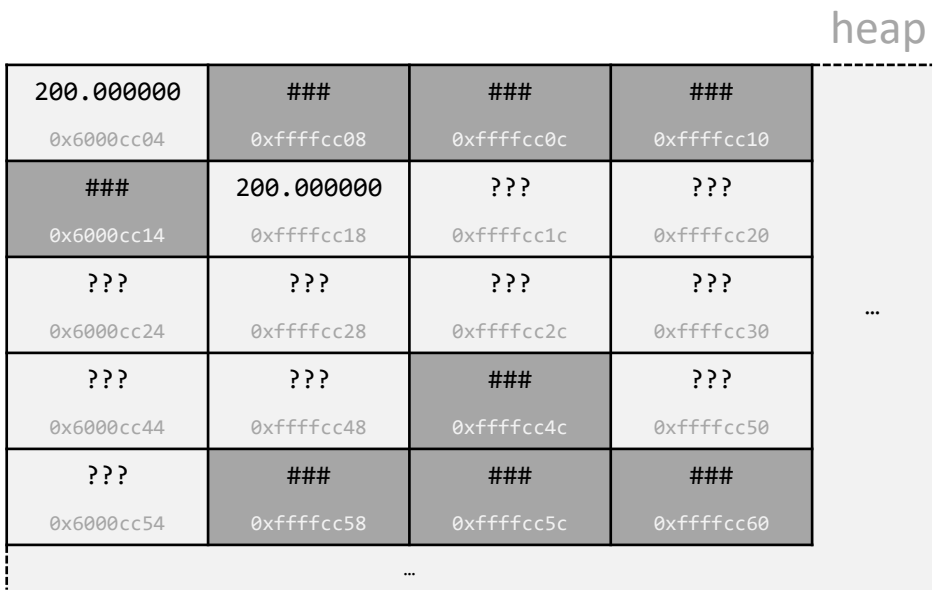


free

2/3

```
…
int main() {
    double *pDHeap1 = (double*) malloc(sizeof(double));
    double *pDHeap2 = (double*) malloc(sizeof(double));

    *pDHeap1 = 200.0;
    *pDHeap2 = 200.0;

    free(pDHeap1);

    pDHeap1 = NULL;

    free(pDHeap2);

    printf("%p %p %lf", pDHeap1, pDHeap2, *pDHeap2);

    return 0;
}
```



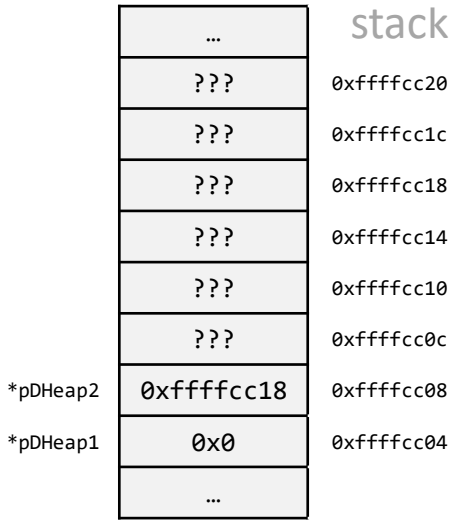


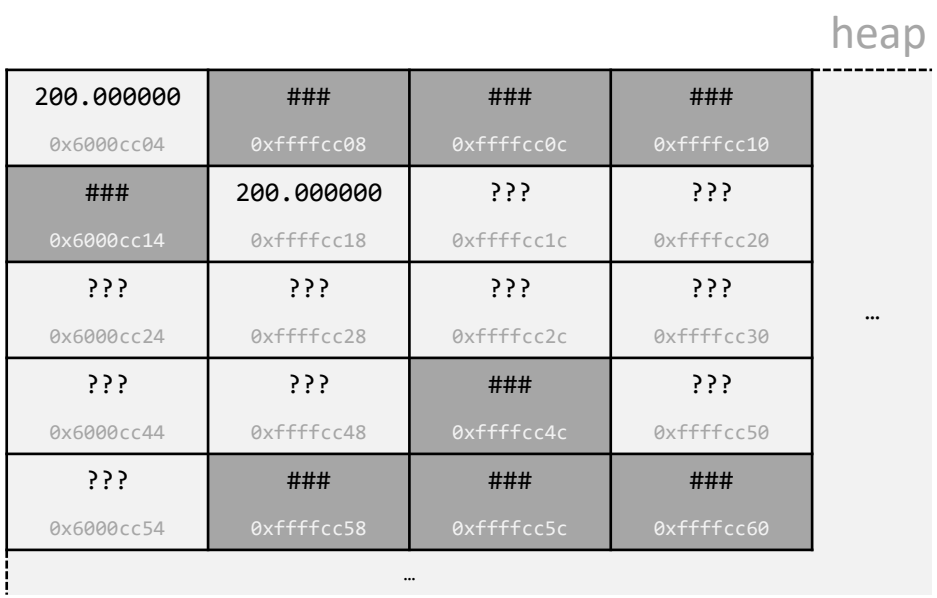
free

3/3

```
…
int main() {
    double *pDHeap1 = (double*) malloc(sizeof(double));
    double *pDHeap2 = (double*) malloc(sizeof(double));

    *pDHeap1 = 200.0;
    *pDHeap2 = 200.0;

    free(pDHeap1);

    pDHeap1 = NULL;

    free(pDHeap2);

    printf("%p %p %lf", pDHeap1, pDHeap2, *pDHeap2);

    return 0;
}
```

```
0x0 0xffffcc18 200.000000
```



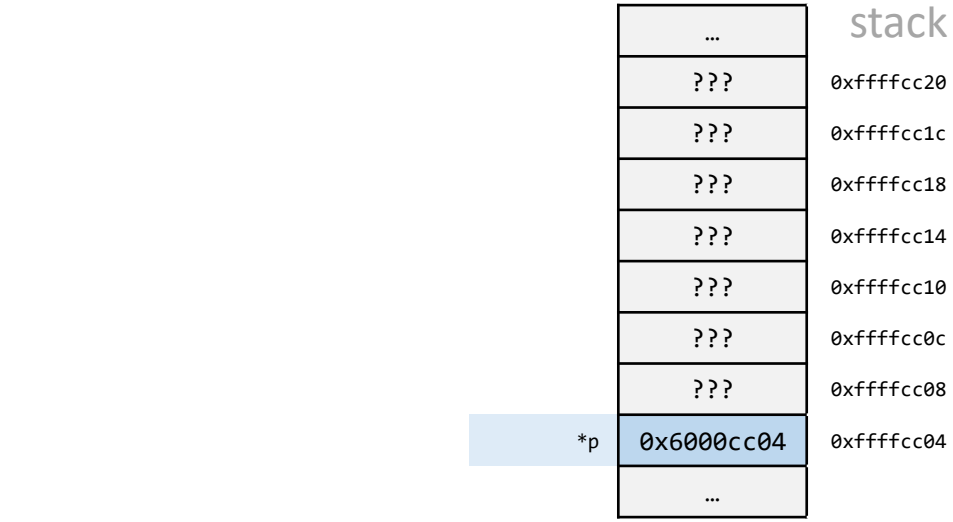


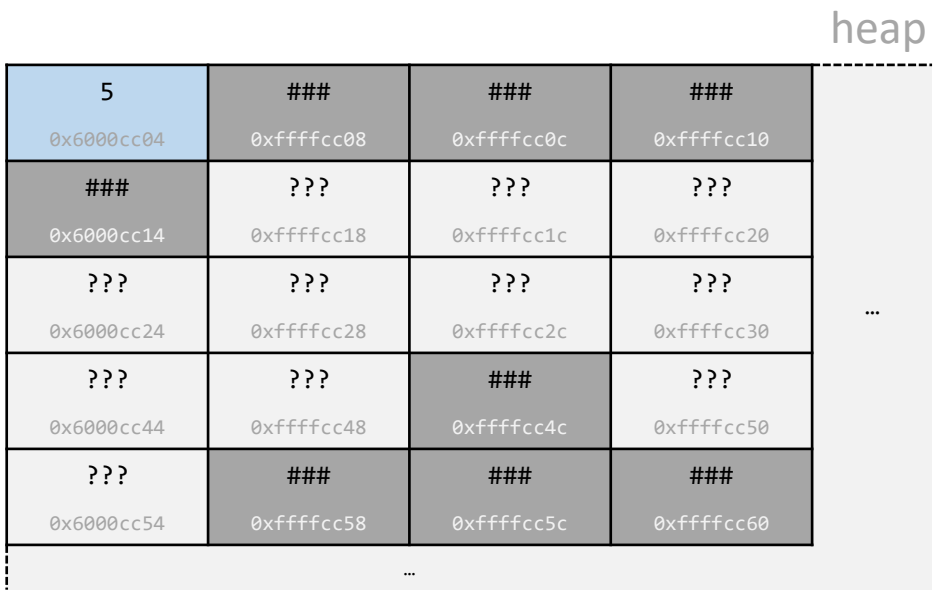
Precauções

Fugas de memória

1/3

```c
#include <stdlib.h>
int main() {
    int *p = (int*) malloc(sizeof(int));
    *p = 5;

    p = (int*) malloc(sizeof(int));
    *p = 8;

    free(p);
    p = NULL;

    return 0;
}
```

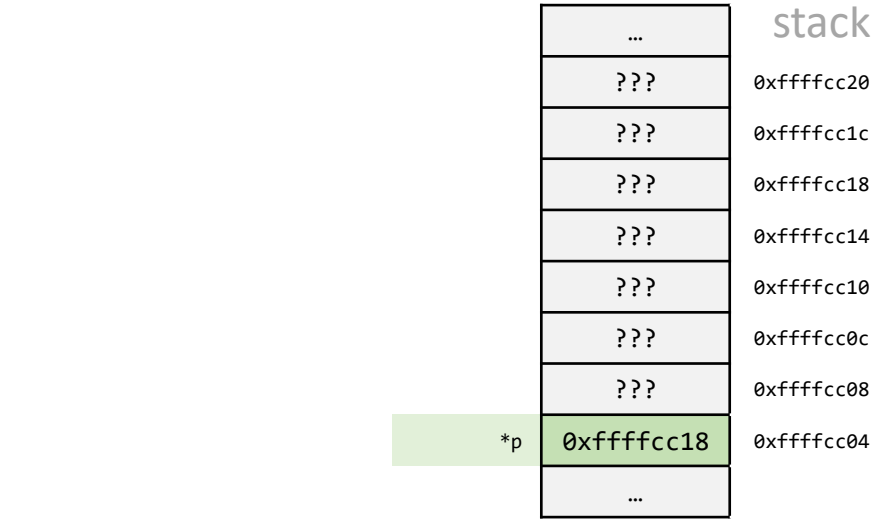


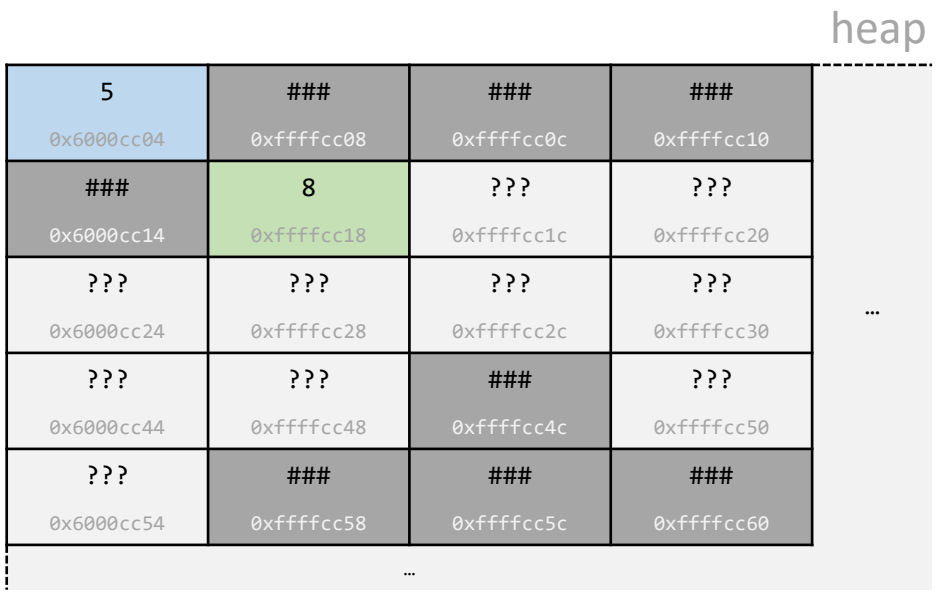


Fugas de memória

2/3

```c
#include <stdlib.h>
int main() {
    int *p = (int*) malloc(sizeof(int));
    *p = 5;

    p = (int*) malloc(sizeof(int));
    *p = 8;

    free(p);
    p = NULL;

    return 0;
}
```

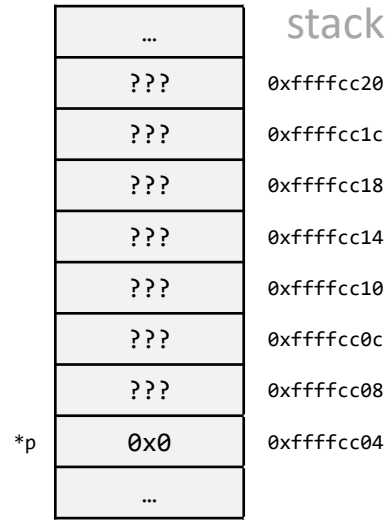




Fugas de memória

3/3

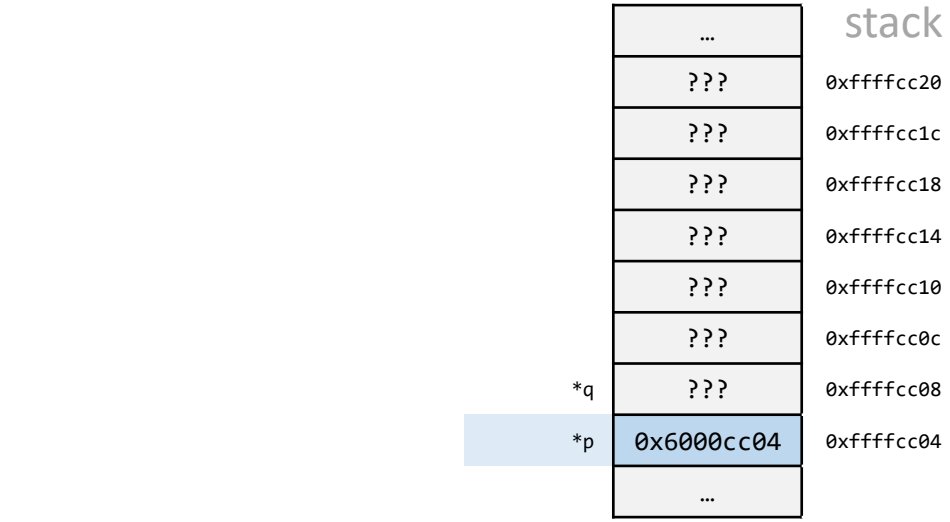
```c
#include <stdlib.h>
int main() {
    int *p = (int*) malloc(sizeof(int));
    *p = 5;

    p = (int*) malloc(sizeof(int));
    *p = 8;

    free(p);
    p = NULL;

    return 0;
}
```

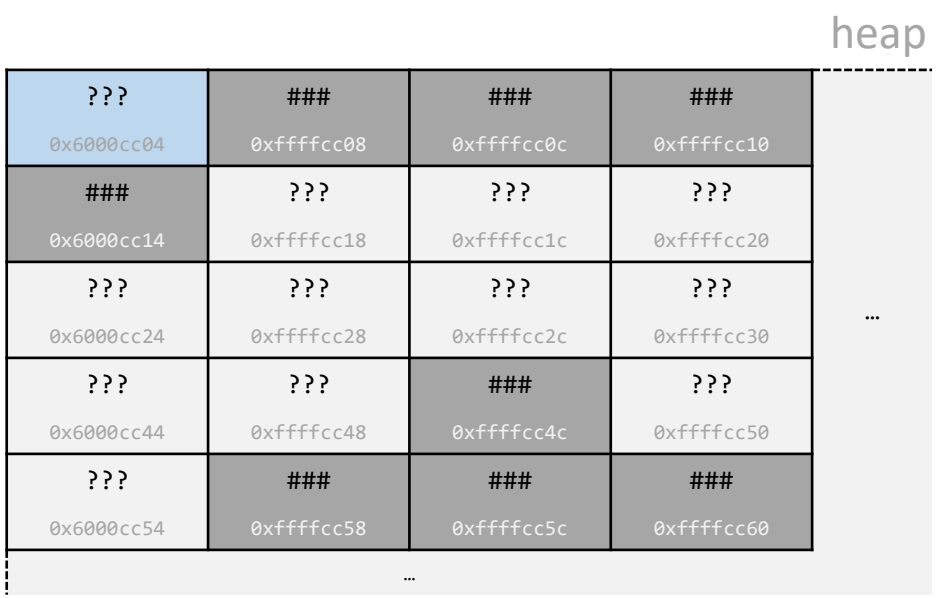


Referências inválidas

1/5

```
…
int *q, *p = (int*) malloc(sizeof(int));

q = p;

*q = 3;

free(q);
q = NULL;

printf("%d %d", *p, *q);
…
```

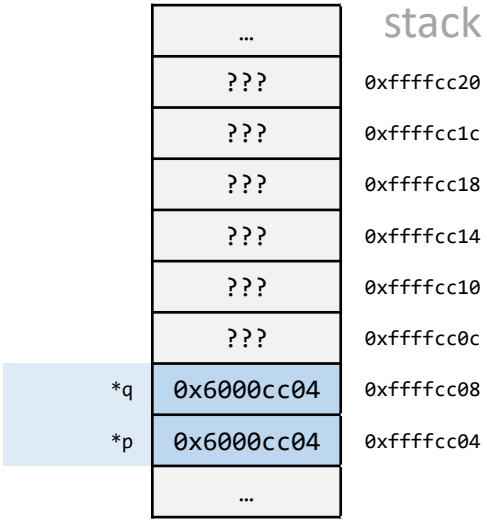


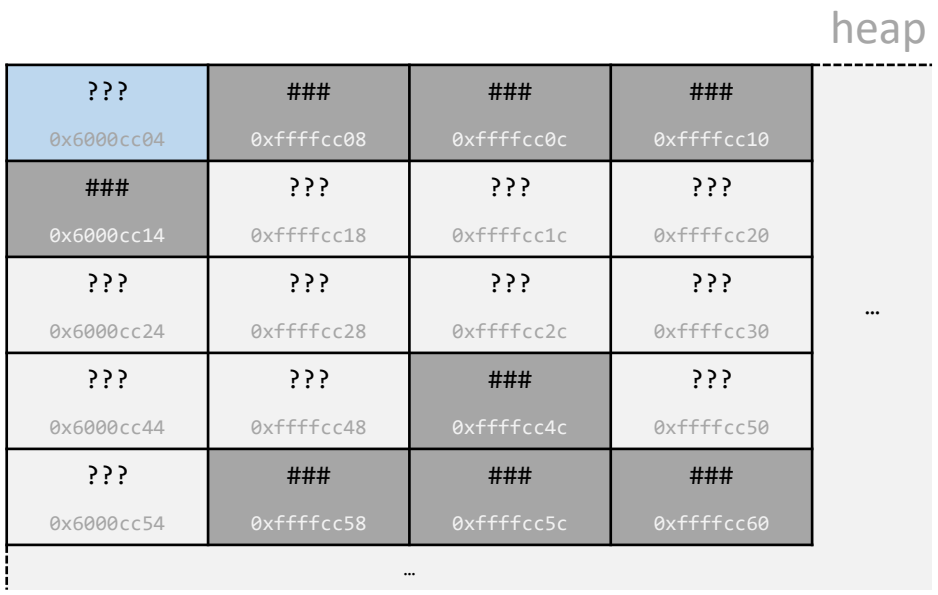


Referências inválidas

2/5

```
…
int *q, *p = (int*) malloc(sizeof(int));

q = p;

*q = 3;

free(q);
q = NULL;

printf("%d %d", *p, *q);
…
```

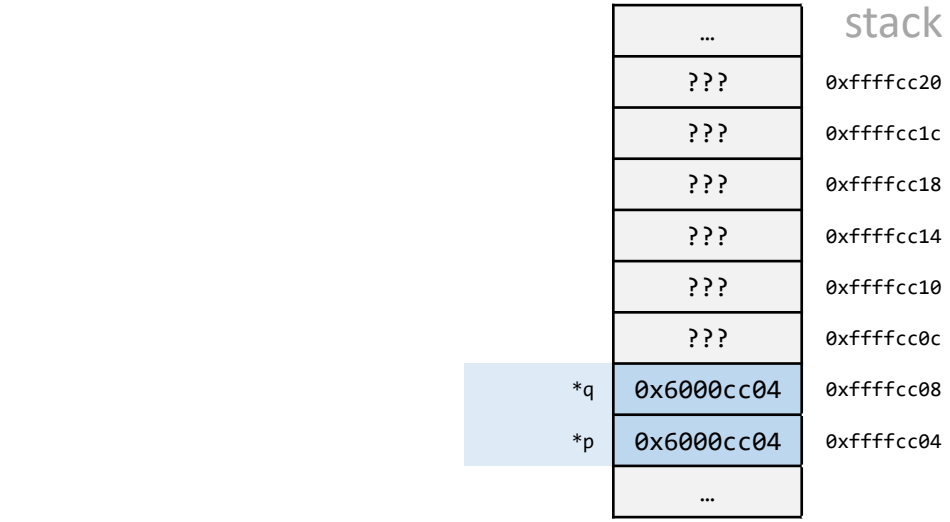


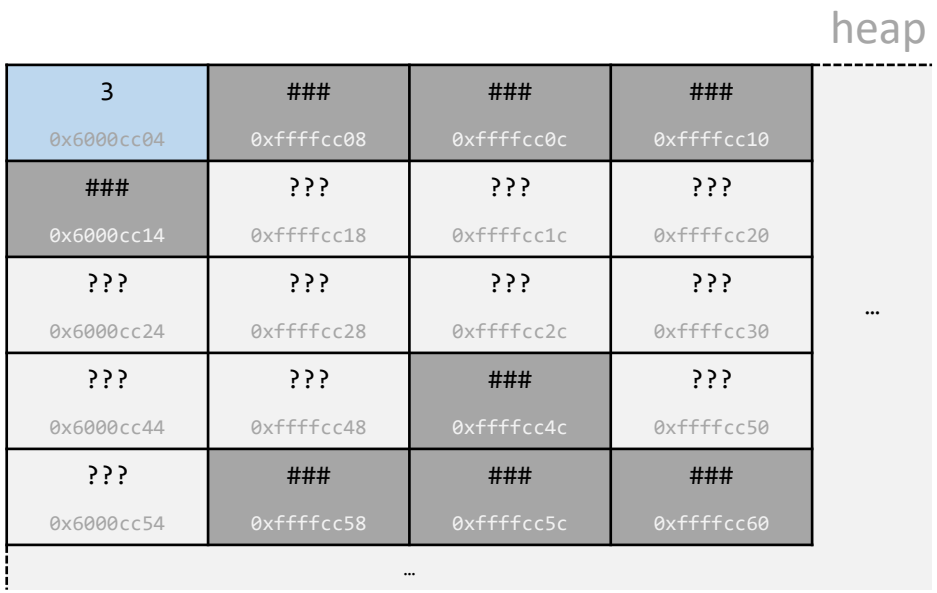


Referências inválidas

3/5

```
…
int *q, *p = (int*) malloc(sizeof(int));

q = p;

*q = 3;

free(q);
q = NULL;

printf("%d %d", *p, *q);
…
```

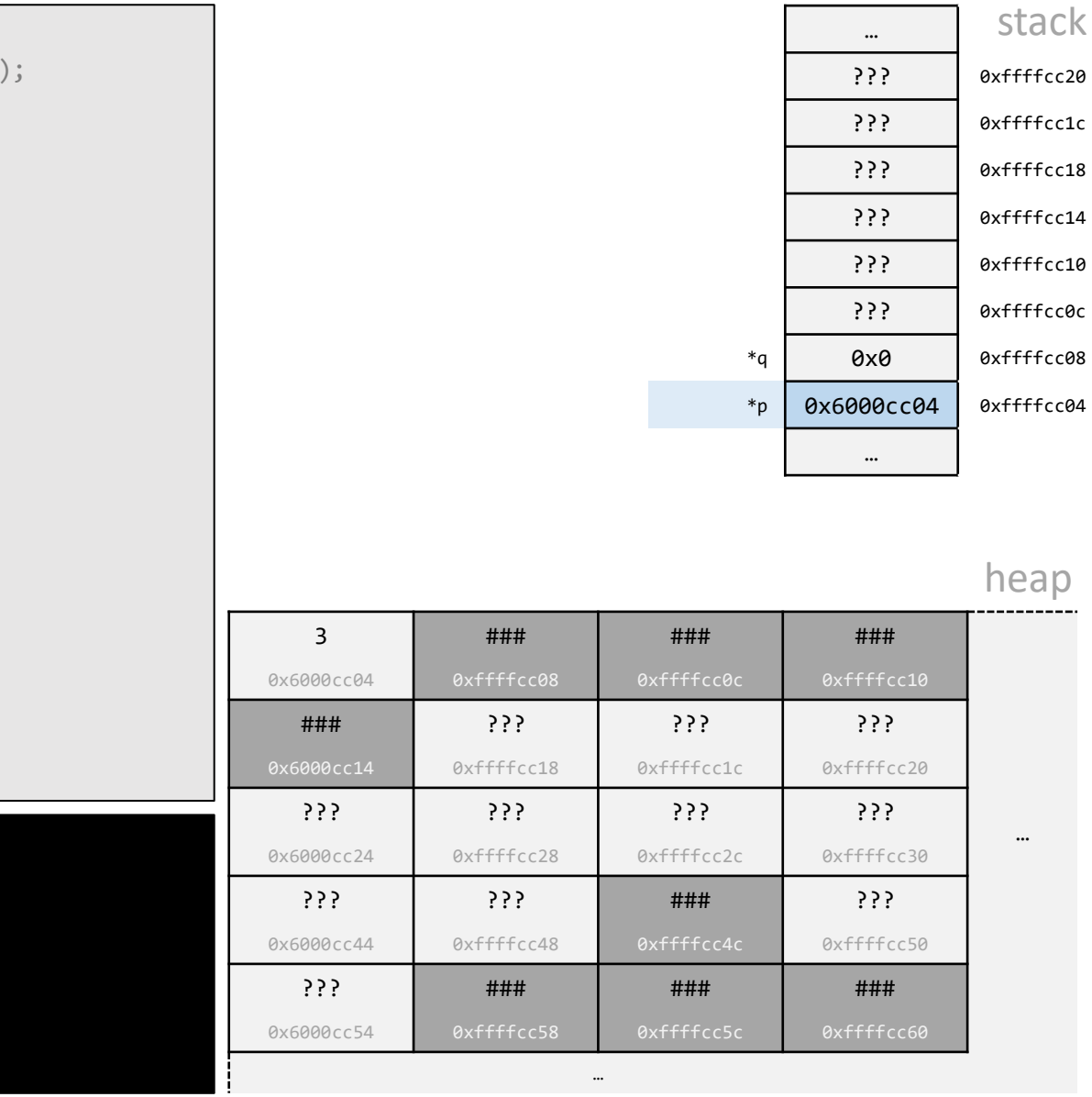




Referências inválidas

4/5

```
…
int *q, *p = (int*) malloc(sizeof(int));

q = p;

*q = 3;

free(q);
q = NULL;

printf("%d %d", *p, *q);
…
```

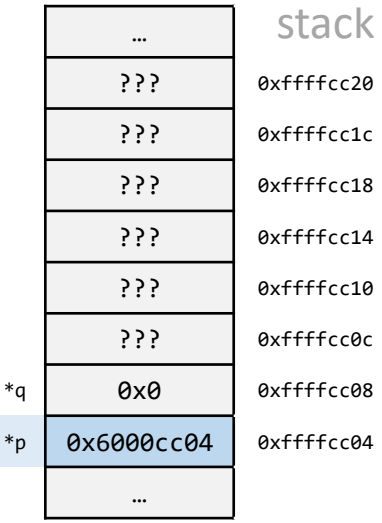


Referências inválidas

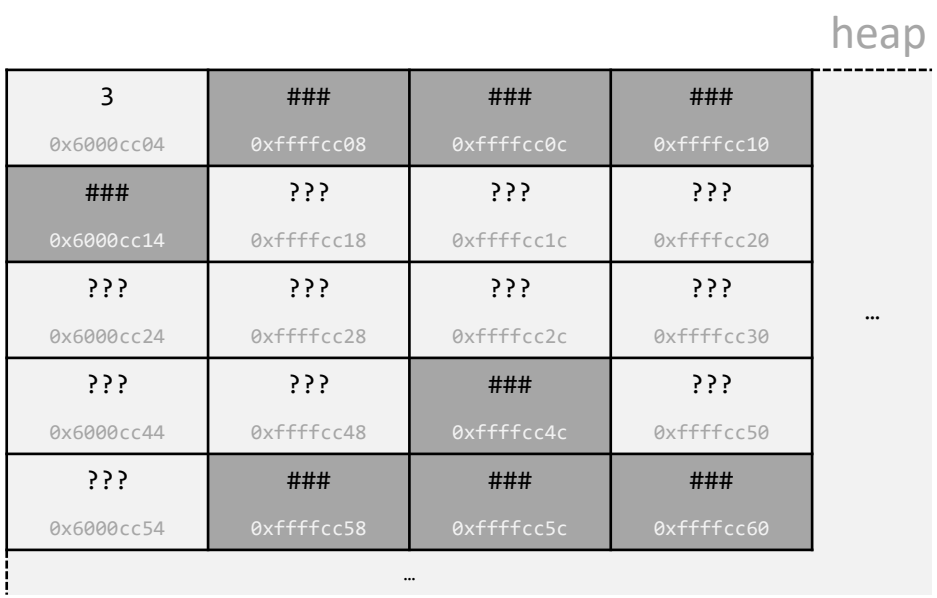
5/5

```
…
int *q, *p = (int*) malloc(sizeof(int));

q = p;

*q = 3;

free(q);
q = NULL;

printf("%d %d", *p, *q);
…
```

q aponta para NULL



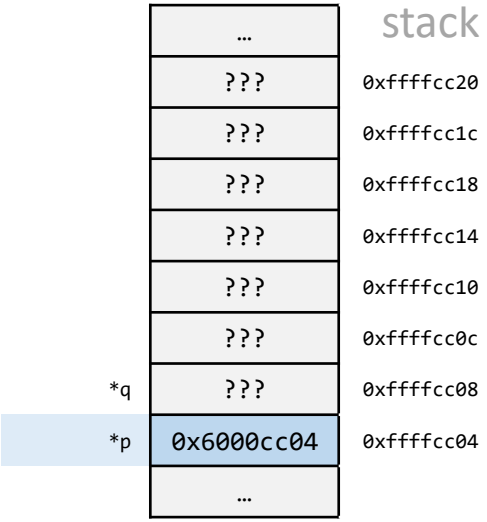

```
RUN FAILED (exit value 1, total time: 100ms)
```

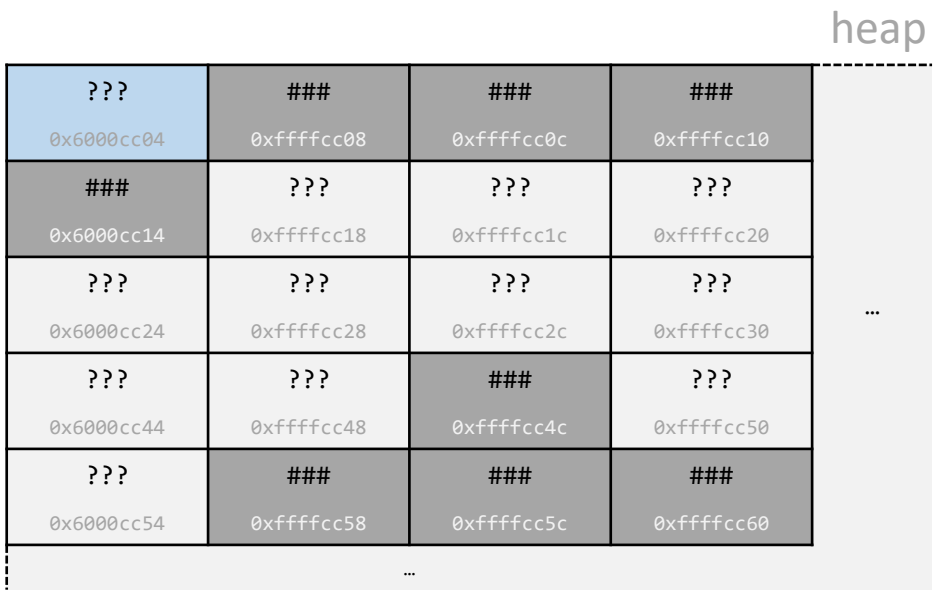


Libertar memória

1/5

```
…
int *q, *p = (int*) malloc(sizeof(int));

q = p;

*q = 3;

free(q);
q = NULL;

free(p);
p = NULL;
…
```

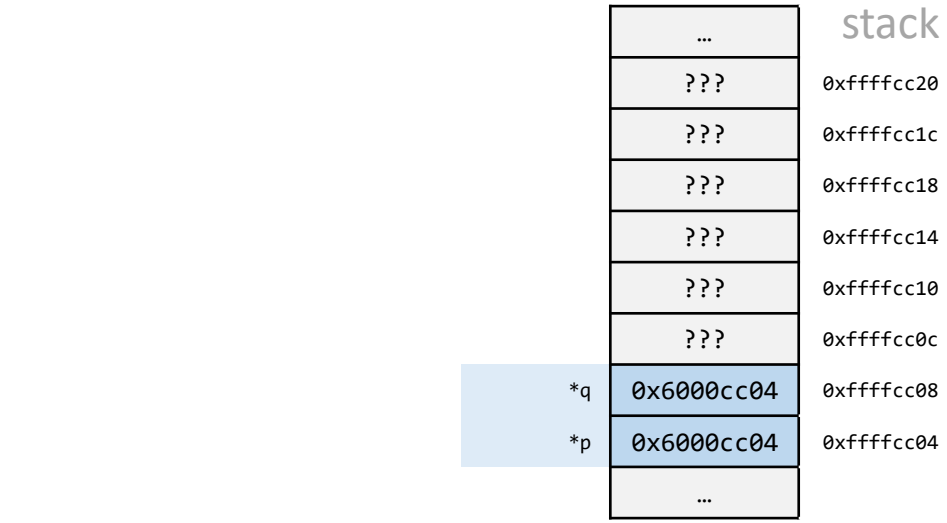


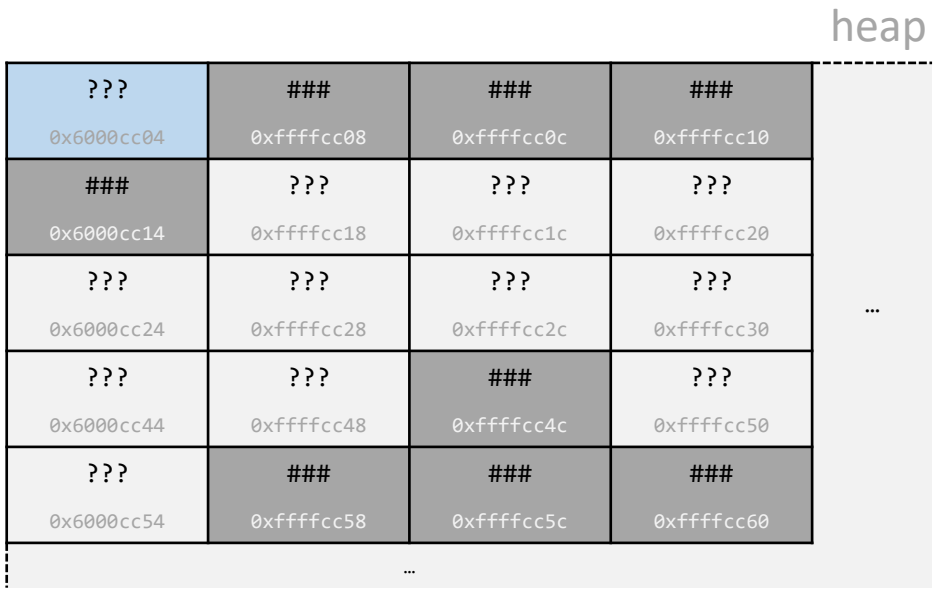


Libertar memória

2/5

```
…
int *q, *p = (int*) malloc(sizeof(int));

q = p;

*q = 3;

free(q);
q = NULL;

free(p);
p = NULL;
…
```

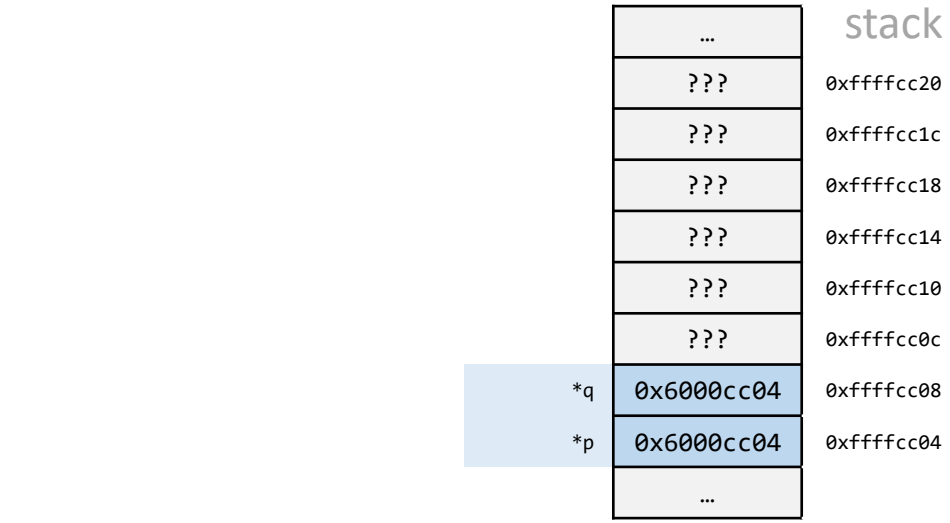


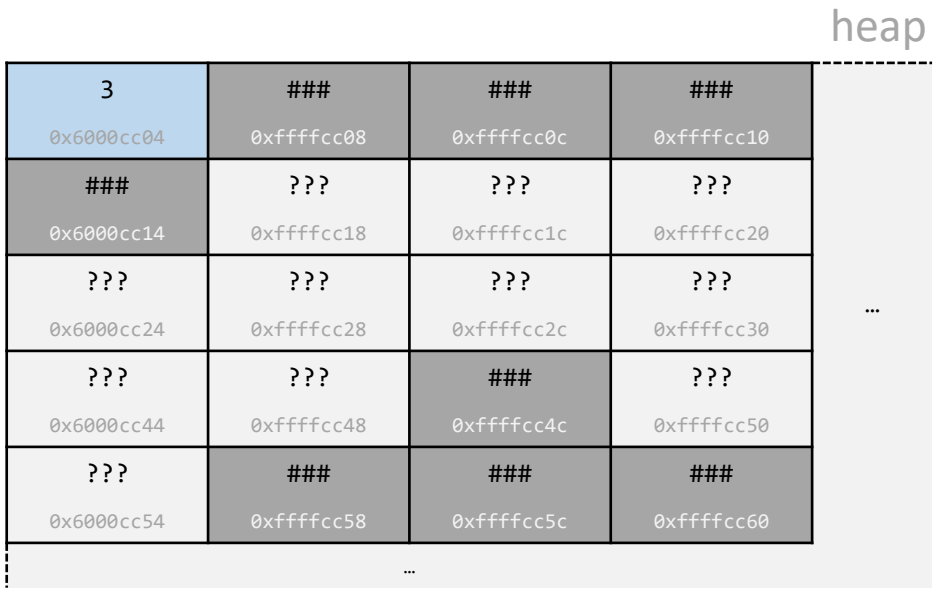


Libertar memória

3/5

```
…
int *q, *p = (int*) malloc(sizeof(int));

q = p;

*q = 3;

free(q);
q = NULL;

free(p);
p = NULL;
…
```

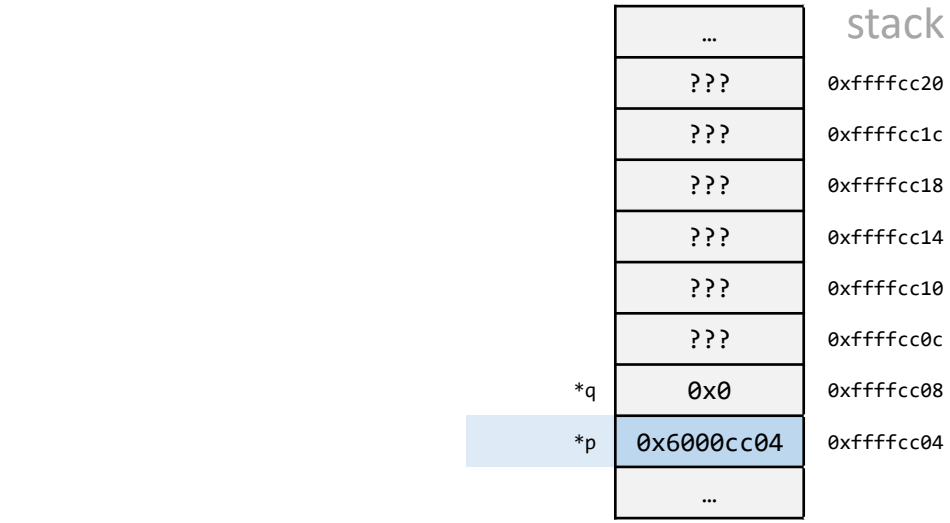


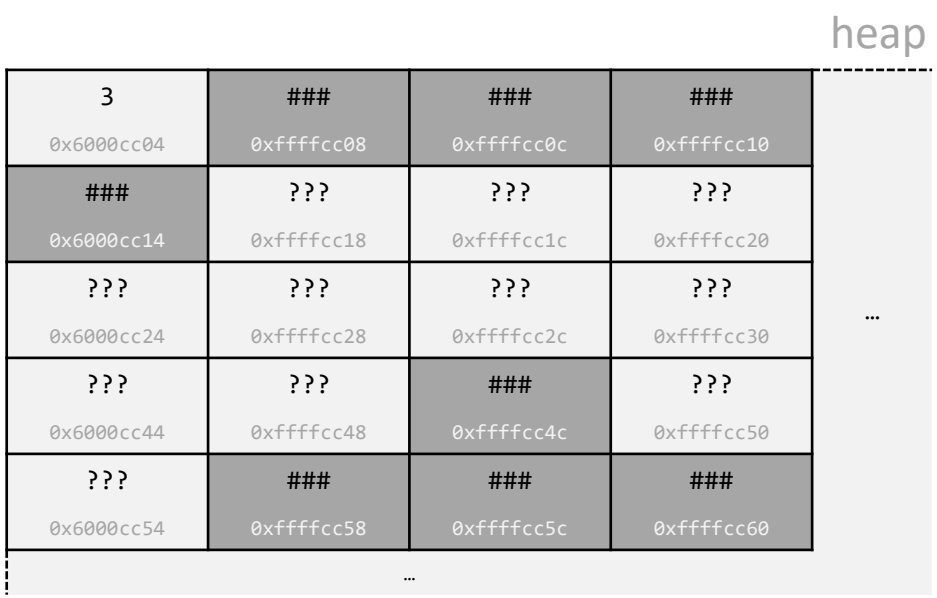


Libertar memória

4/5

```
…
int *q, *p = (int*) malloc(sizeof(int));

q = p;

*q = 3;

free(q);
q = NULL;

free(p);
p = NULL;
…
```



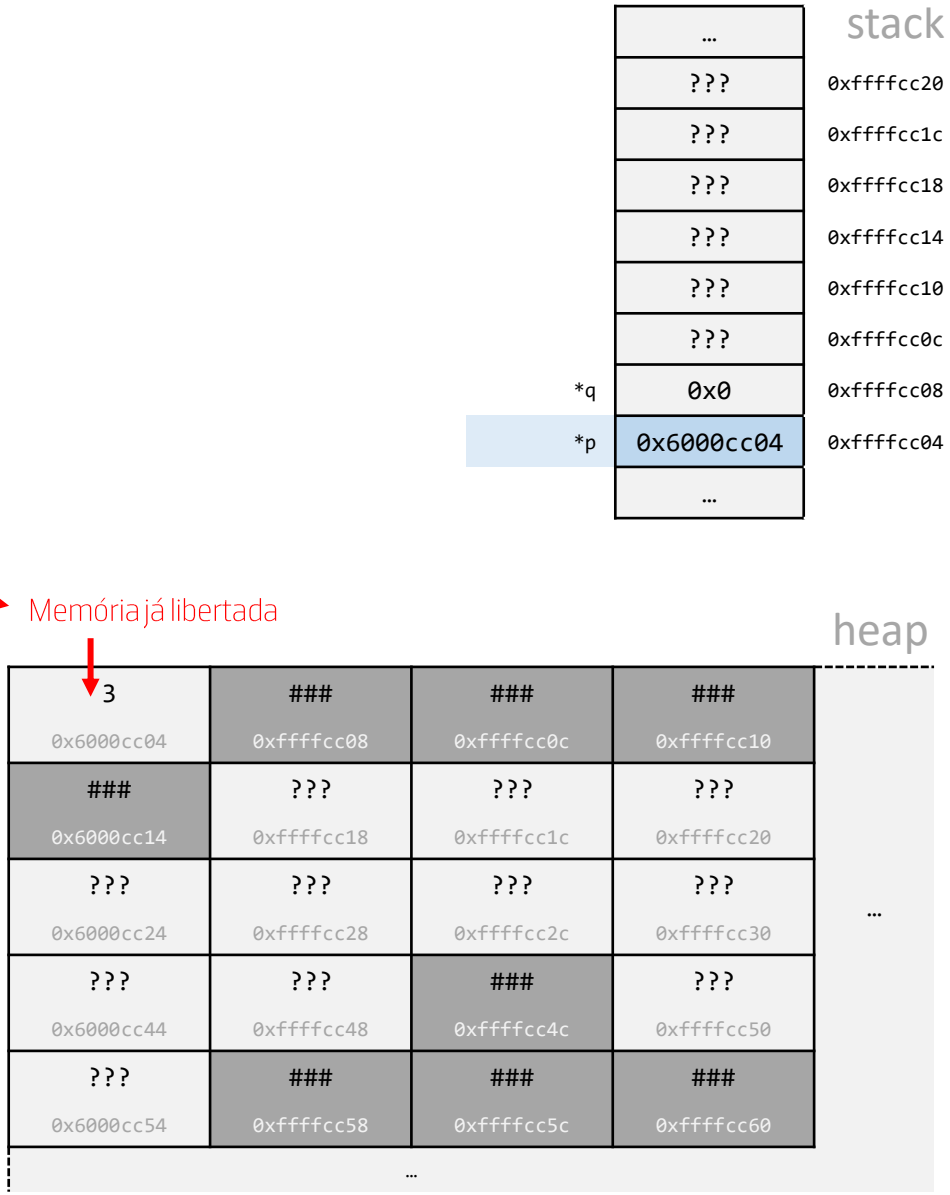


Libertar memória

5/5

```
…
int *q, *p = (int*) malloc(sizeof(int));

q = p;

*q = 3;

free(q);
q = NULL;

free(p);
p = NULL;
…
```

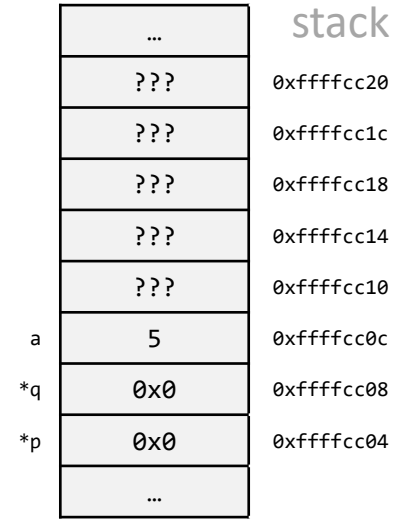
```
RUN FAILED (exit value 1, total time: 147ms)
```

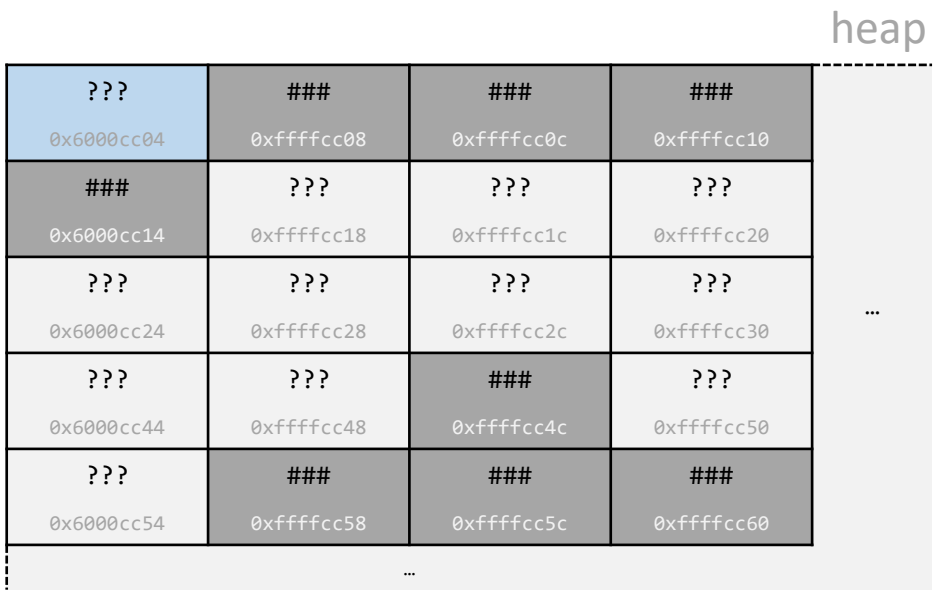


Não libertar variáveis estáticas

1/5

```
…
int *p = NULL, *q = NULL, a = 5;

p = (int*) malloc(sizeof (int));
*p = 3;

q = &a;

free(p);
p = NULL;

free(q);
q = NULL;
…
```

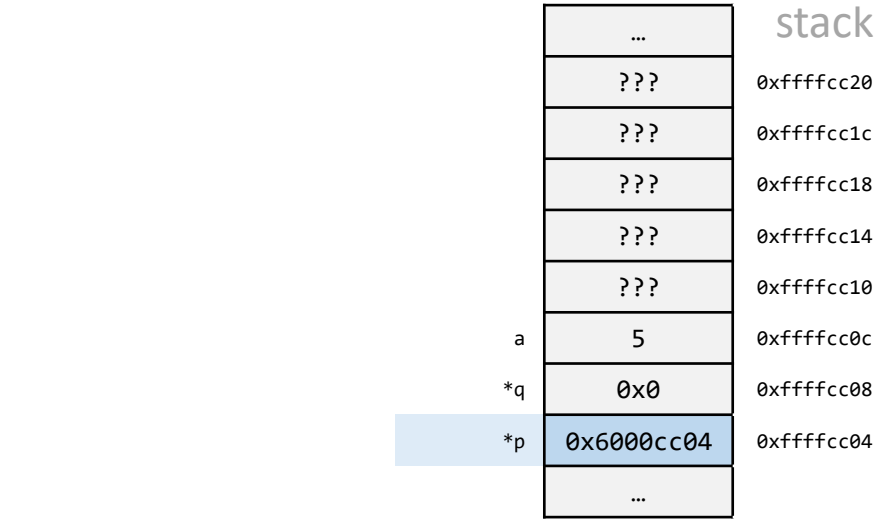


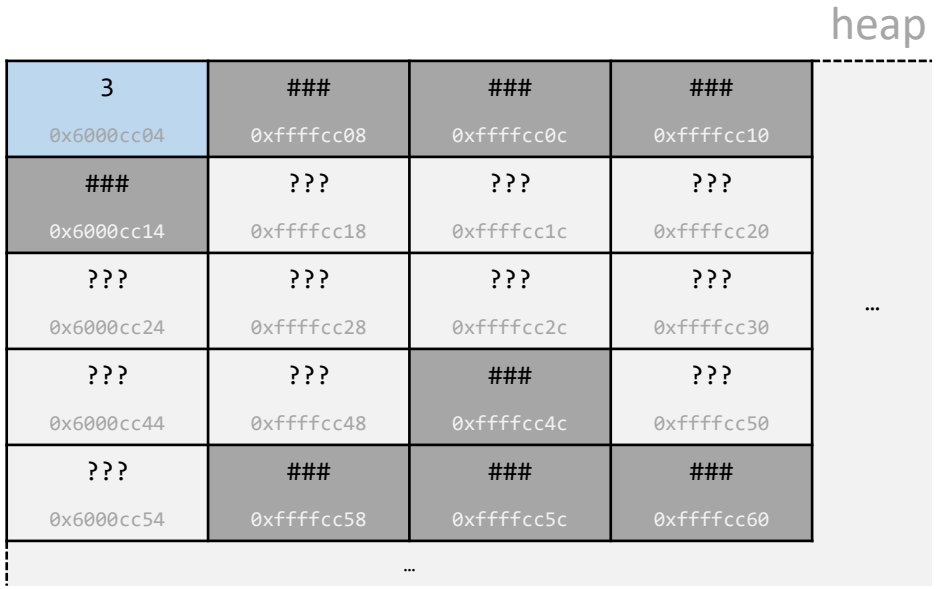


Não libertar variáveis estáticas

2/5

```
…
int *p = NULL, *q = NULL, a = 5;

p = (int*) malloc(sizeof (int));
*p = 3;

q = &a;

free(p);
p = NULL;

free(q);
q = NULL;
…
```

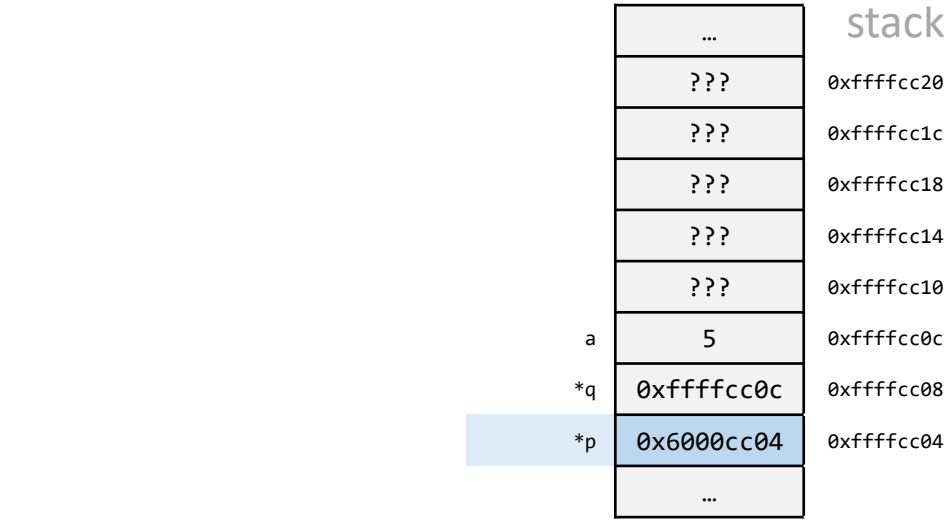


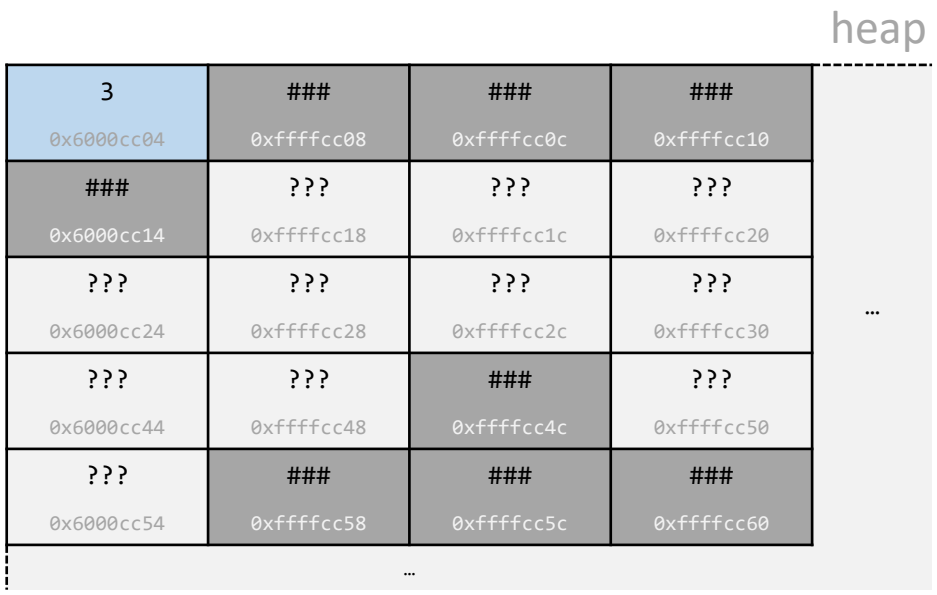


Não libertar variáveis estáticas

3/5

```
…
int *p = NULL, *q = NULL, a = 5;

p = (int*) malloc(sizeof (int));
*p = 3;

q = &a;

free(p);
p = NULL;

free(q);
q = NULL;
…
```

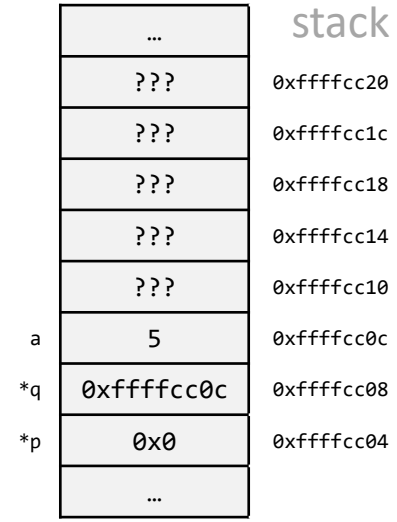


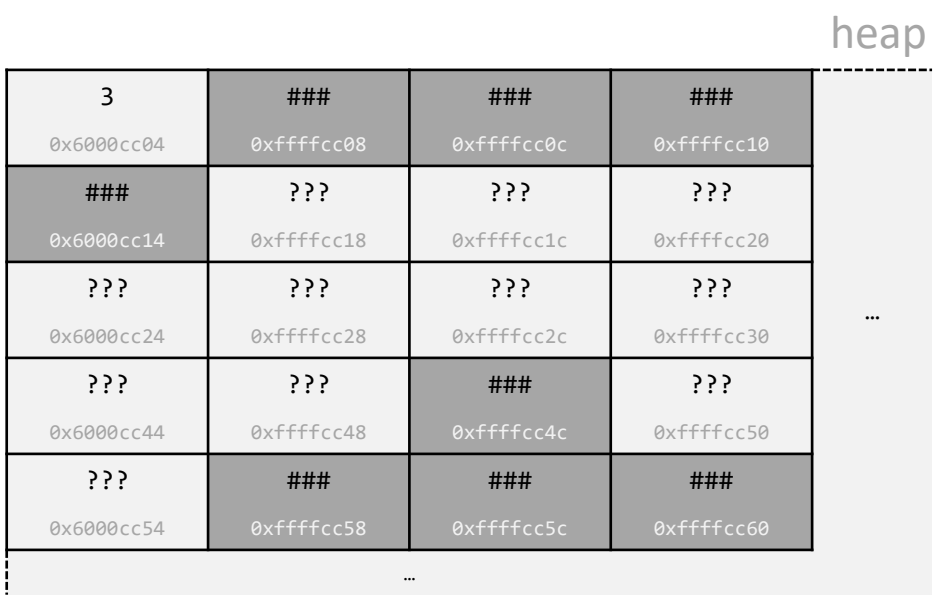


Não libertar variáveis estáticas

4/5

```
…
int *p = NULL, *q = NULL, a = 5;

p = (int*) malloc(sizeof (int));
*p = 3;

q = &a;

free(p);
p = NULL;

free(q);
q = NULL;
…
```



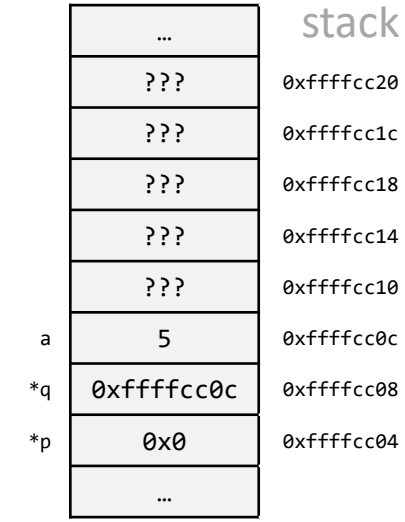


Não libertar variáveis estáticas

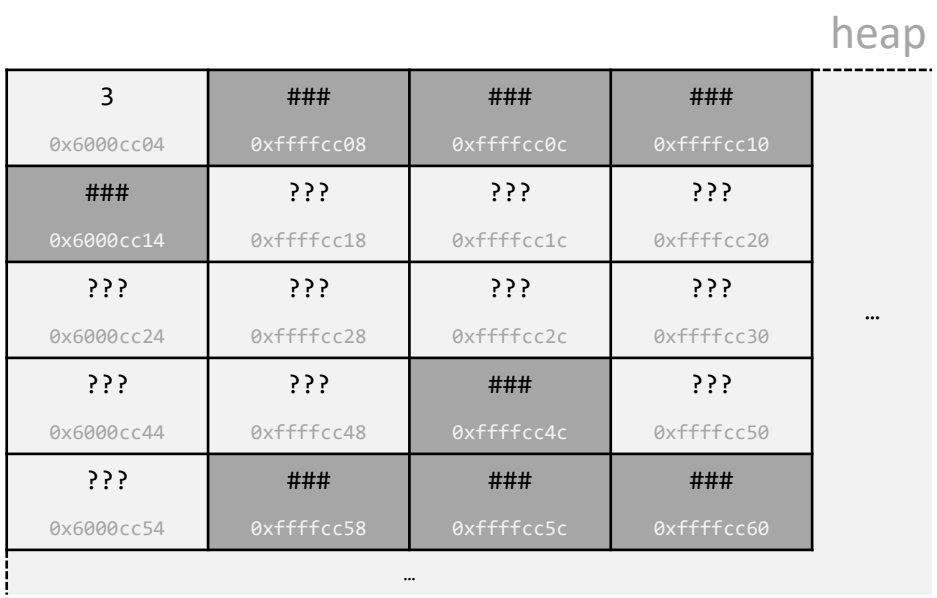
5/5

```
…
int *p = NULL, *q = NULL, a = 5;

p = (int*) malloc(sizeof (int));
*p = 3;

q = &a;

free(p);
p = NULL;

free(q);
q = NULL;
…
```

Tentativa de libertação de memória estática.



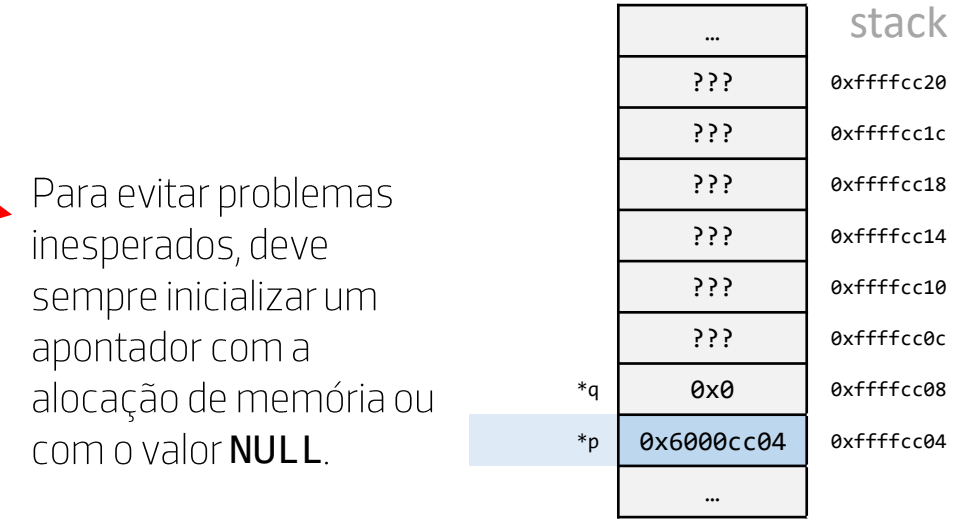

```
RUN FAILED (exit value 1, total time: 144ms)
```

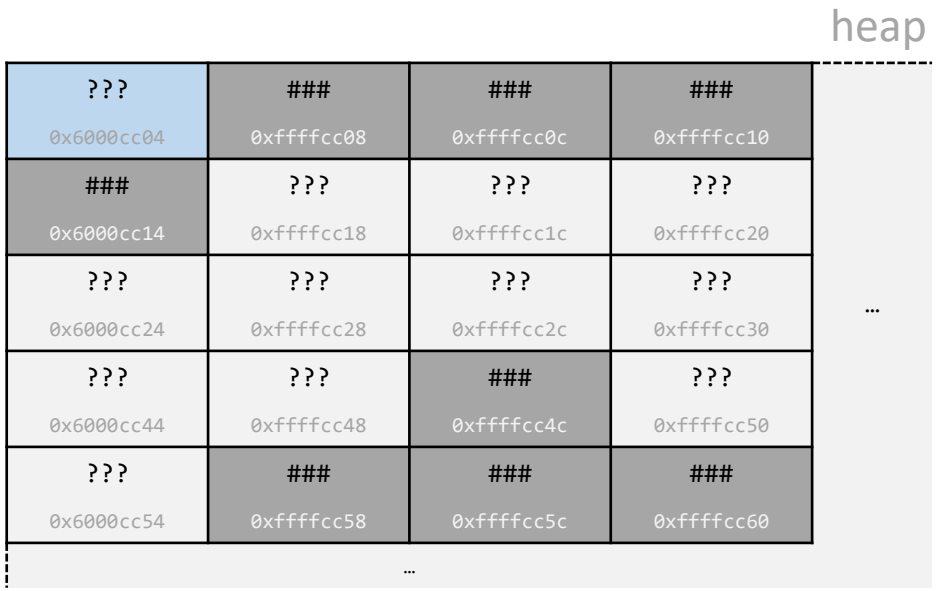


Não libertar apontadores indefinidos

```
…
int *p = (int*) malloc(sizeof(int))
int *q = NULL;

free(p);
p = NULL;

free(q);
q = NULL;
…
```

Para evitar problemas inesperados, deve sempre inicializar um apontador com a alocação de memória ou com o valor **NULL**.



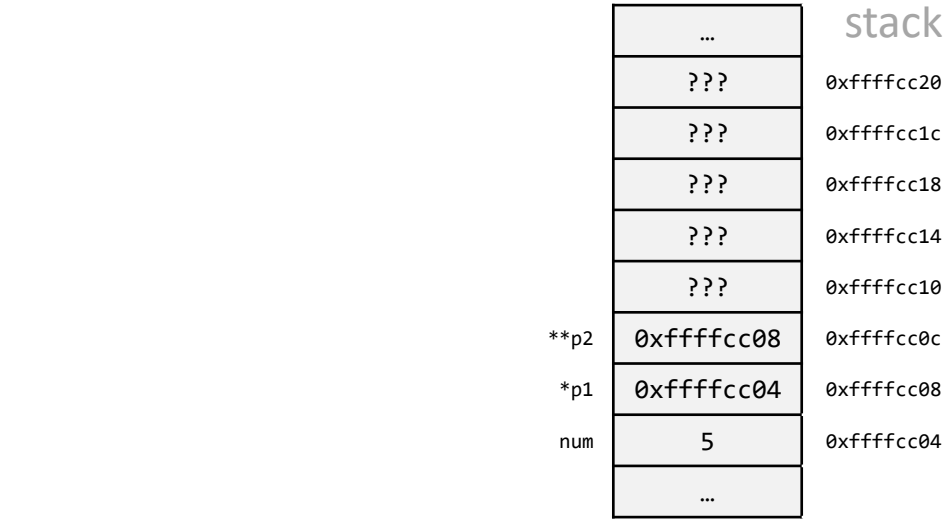


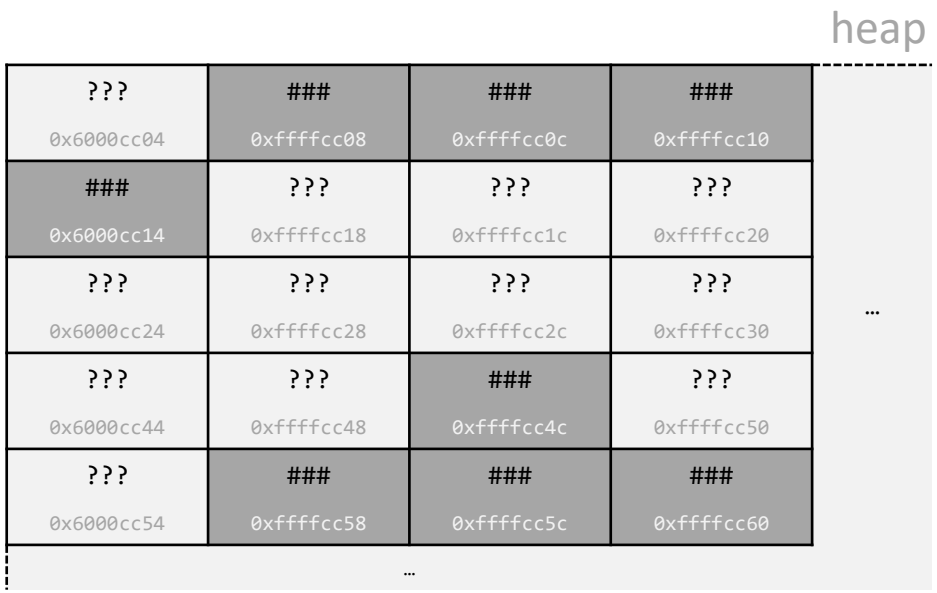
Apontadores para apontadores

Apontadores para Apontadores

1/4

```
…
int num = 5;
int *p1 = &num;
int **p2 = &p1;

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

printf("%p %p %d\n", &p1, p1, *p1);

printf("\n%p %p %p %d", &p2, p2, *p2, **p2);
…
```



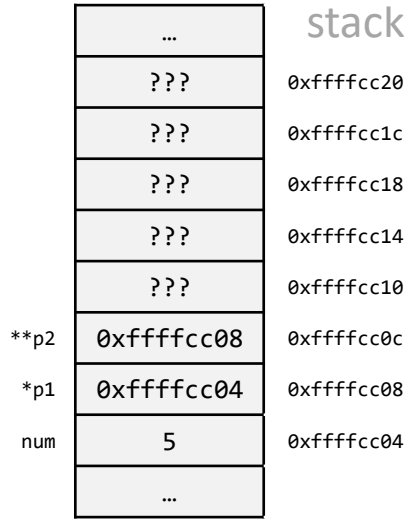


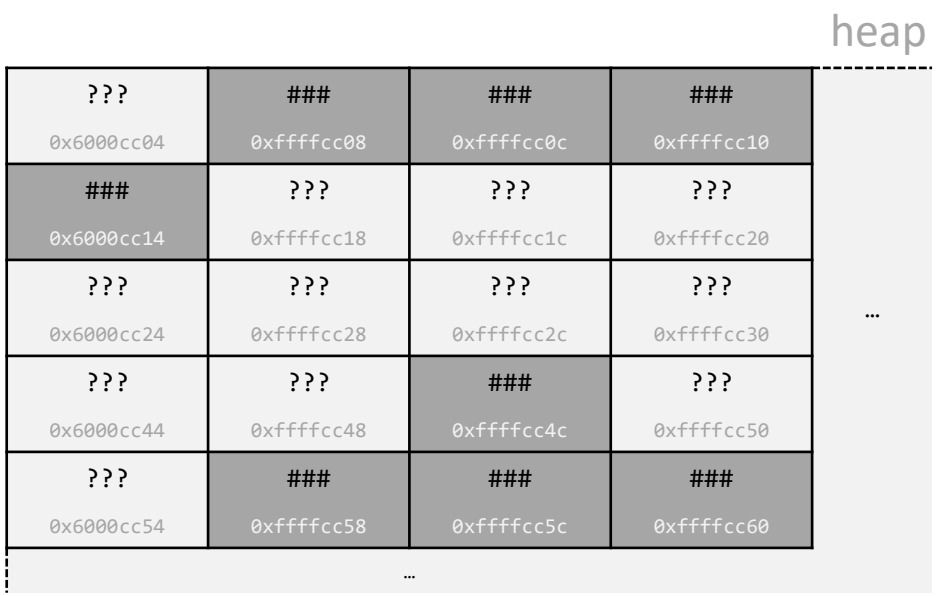
Apontadores para Apontadores

2/4

```
…
int num = 5;
int *p1 = &num;
int **p2 = &p1;

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

printf("%p %p %d\n", &p1, p1, *p1);

printf("\n%p %p %p %d", &p2, p2, *p2, **p2);
…
```

```
0xffffcc04 5
```



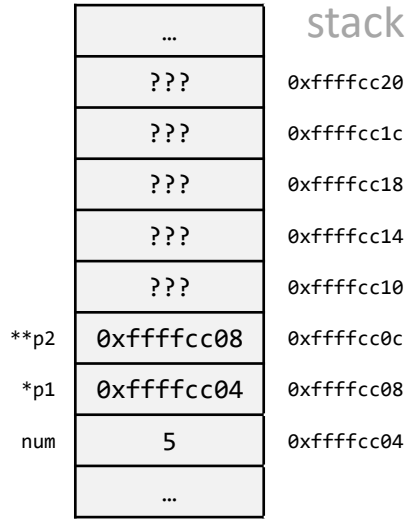


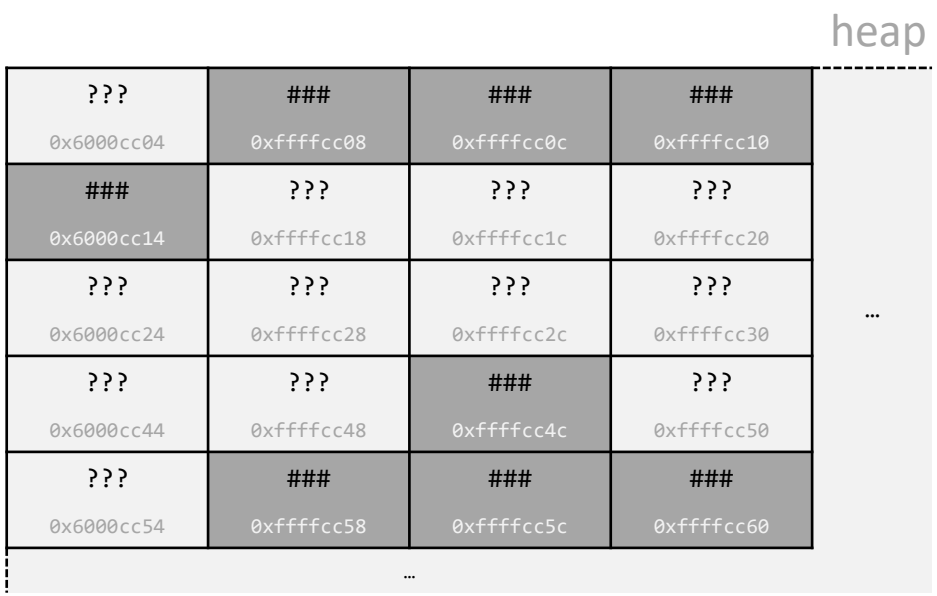
Apontadores para Apontadores

3/4

```
…
int num = 5;
int *p1 = &num;
int **p2 = &p1;

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

printf("%p %p %d\n", &p1, p1, *p1);

printf("\n%p %p %p %d", &p2, p2, *p2, **p2);
…
```

```
0xffffcc04 5
0xffffcc08 0xffffcc04 5
```



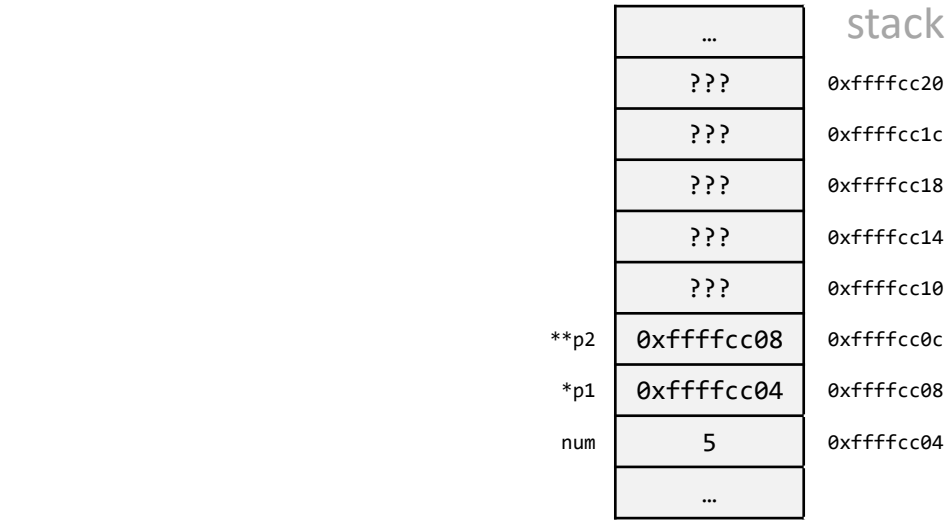


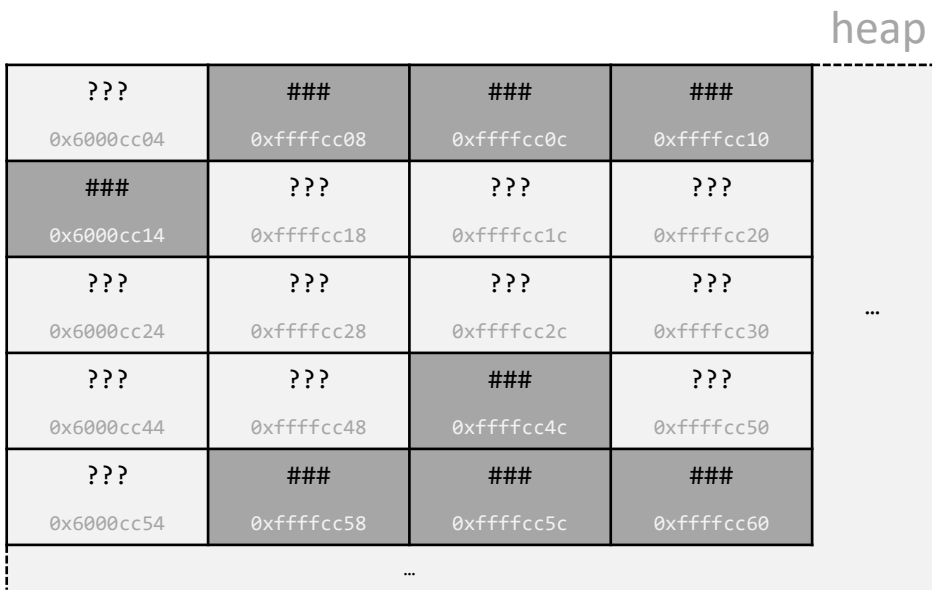
Apontadores para Apontadores

4/4

```
…
int num = 5;
int *p1 = &num;
int **p2 = &p1;

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

printf("%p %p %d\n", &p1, p1, *p1);

printf("\n%p %p %p %d", &p2, p2, *p2, **p2);
…
```

```
0xffffcc04 5
0xffffcc08 0xffffcc04 5
0xffffcc0c 0xffffcc08 0xffffcc04 5
```



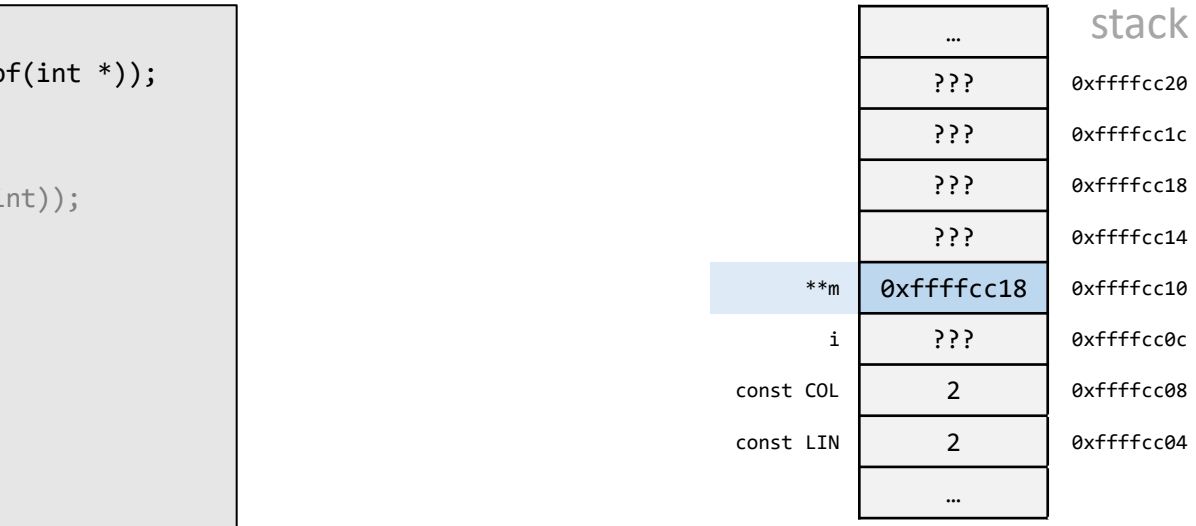


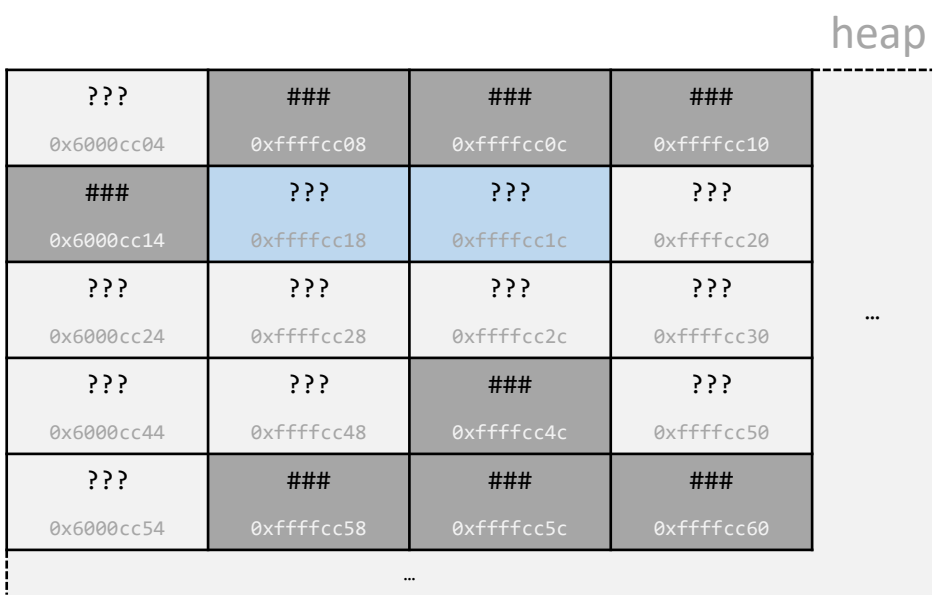
Matrizes

Matrizes

1/5

```
…
int i, **m = (int **) malloc(LIN * sizeof(int *));

for (i = 0; i < LIN; ++i) {
    m[i] = (int *) malloc(COL * sizeof(int));
}

m[0][0] = 10;
printf("%d", m[0][0]);

for (i = 0; i < LIN; ++i) {
    free(m[i]);
    m[i] = NULL;
}

free(m);
m = NULL;
…
```

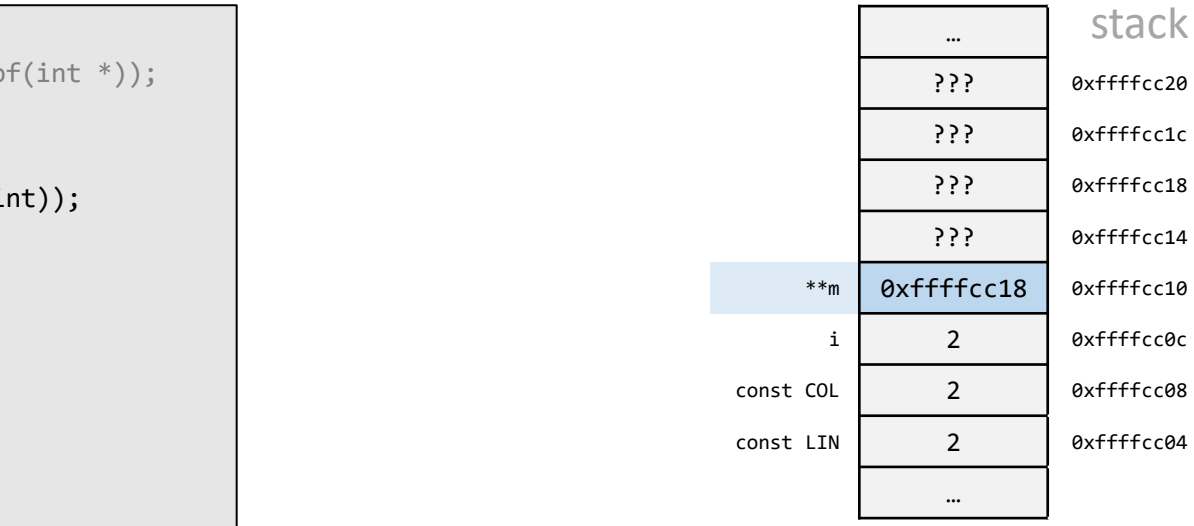


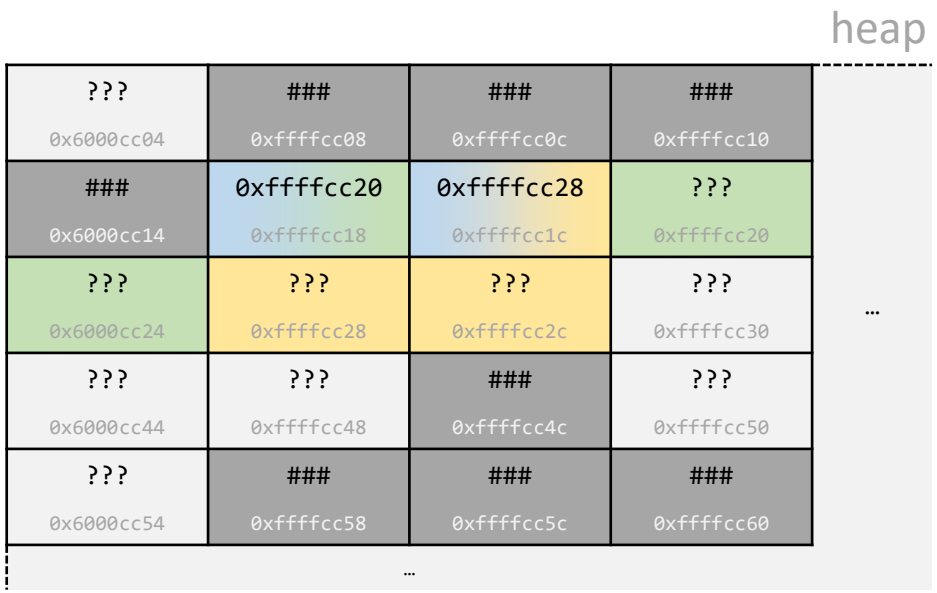


Matrizes

2/5

```
…
int i, **m = (int **) malloc(LIN * sizeof(int *));

for (i = 0; i < LIN; ++i) {
    m[i] = (int *) malloc(COL * sizeof(int));
}

m[0][0] = 10;
printf("%d", m[0][0]);

for (i = 0; i < LIN; ++i) {
    free(m[i]);
    m[i] = NULL;
}

free(m);
m = NULL;
…
```



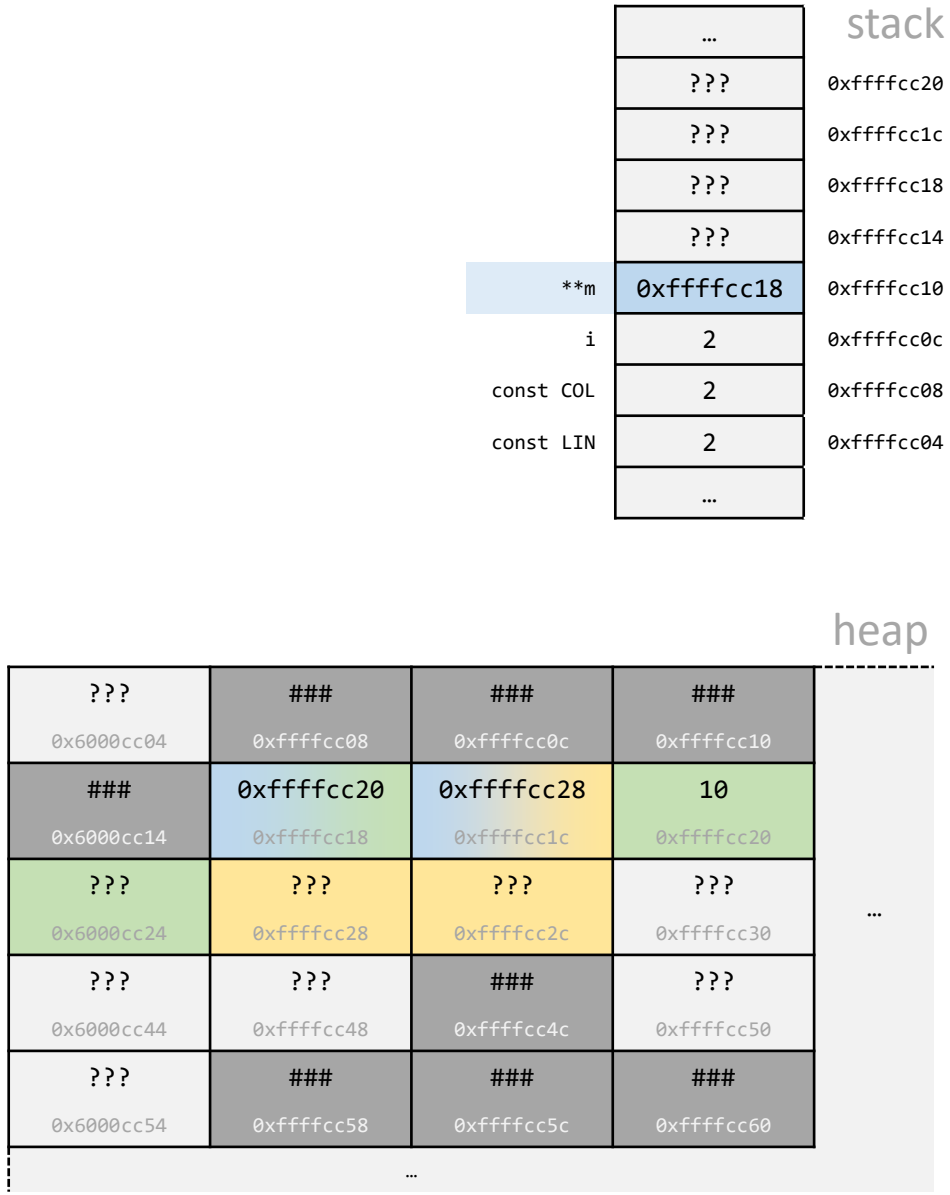


Matrizes

3/5

```
…
int i, **m = (int **) malloc(LIN * sizeof(int *));

for (i = 0; i < LIN; ++i) {
    m[i] = (int *) malloc(COL * sizeof(int));
}

m[0][0] = 10;
printf("%d", m[0][0]);

for (i = 0; i < LIN; ++i) {
    free(m[i]);
    m[i] = NULL;
}

free(m);
m = NULL;
…
```

```
10
```

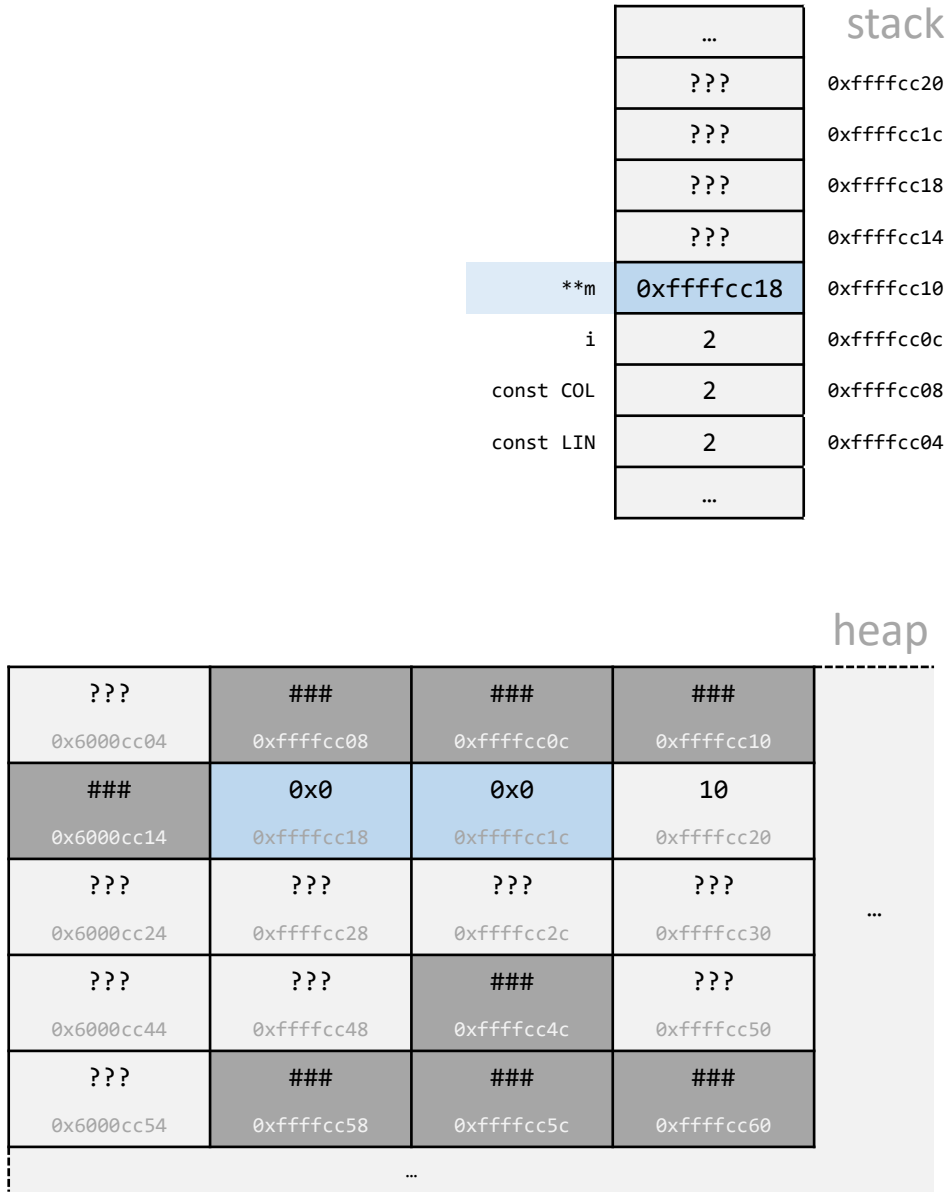


Matrizes

4/5

```
…
int i, **m = (int **) malloc(LIN * sizeof(int *));

for (i = 0; i < LIN; ++i) {
    m[i] = (int *) malloc(COL * sizeof(int));
}

m[0][0] = 10;
printf("%d", m[0][0]);

for (i = 0; i < LIN; ++i) {
    free(m[i]);
    m[i] = NULL;
}

free(m);
m = NULL;
…
```

```
10
```

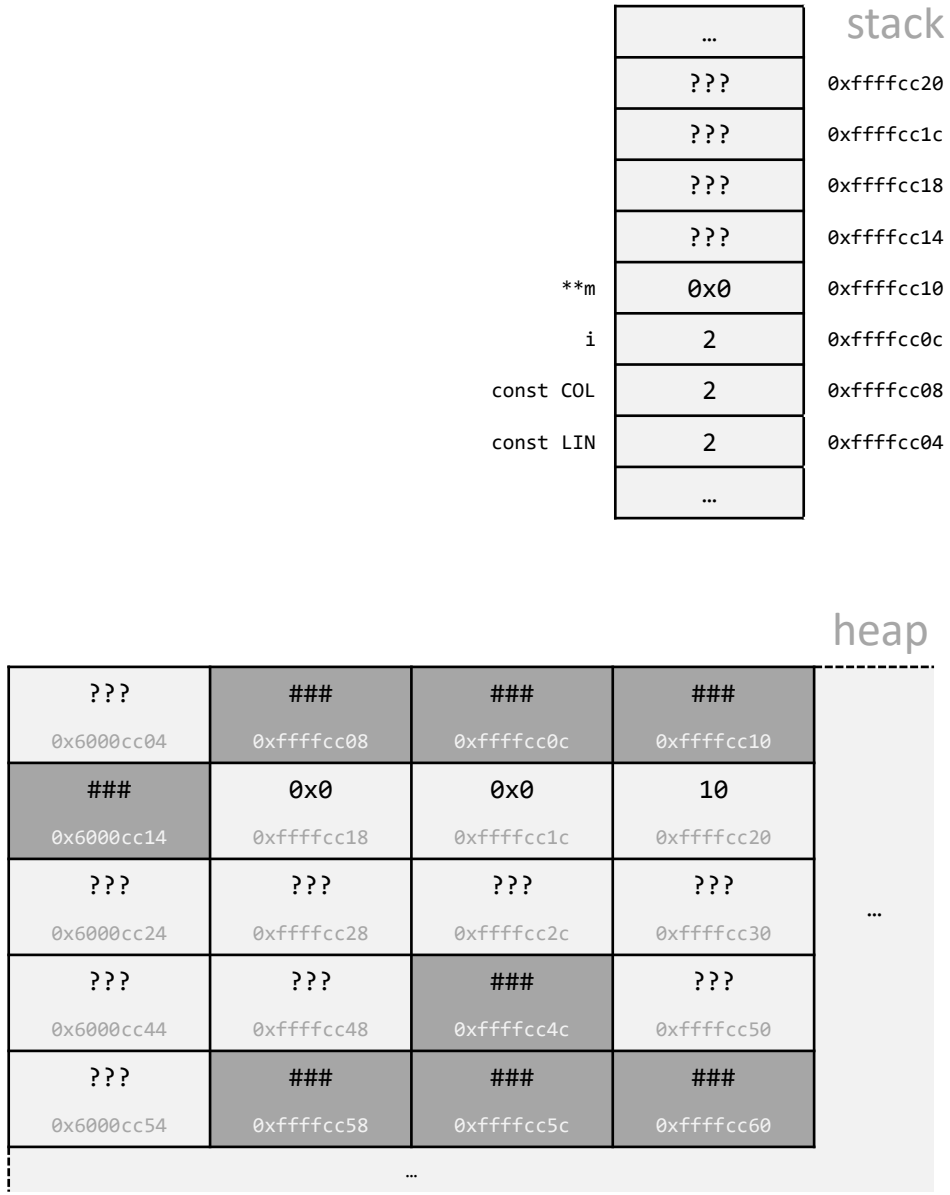


Matrizes

5/5

```
…
int i, **m = (int **) malloc(LIN * sizeof(int *));

for (i = 0; i < LIN; ++i) {
    m[i] = (int *) malloc(COL * sizeof(int));
}

m[0][0] = 10;
printf("%d", m[0][0]);

for (i = 0; i < LIN; ++i) {
    free(m[i]);
    m[i] = NULL;
}

free(m);
m = NULL;
…
```

```
10
```



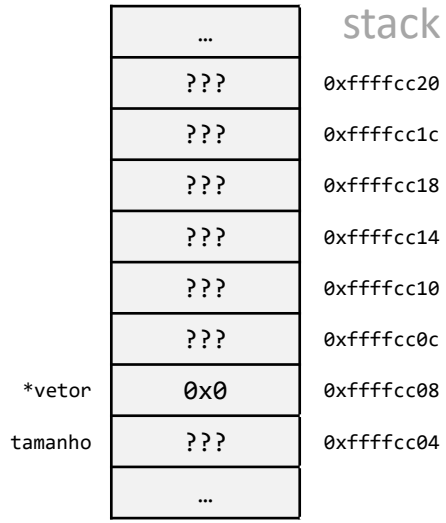
Alocar memória em funções

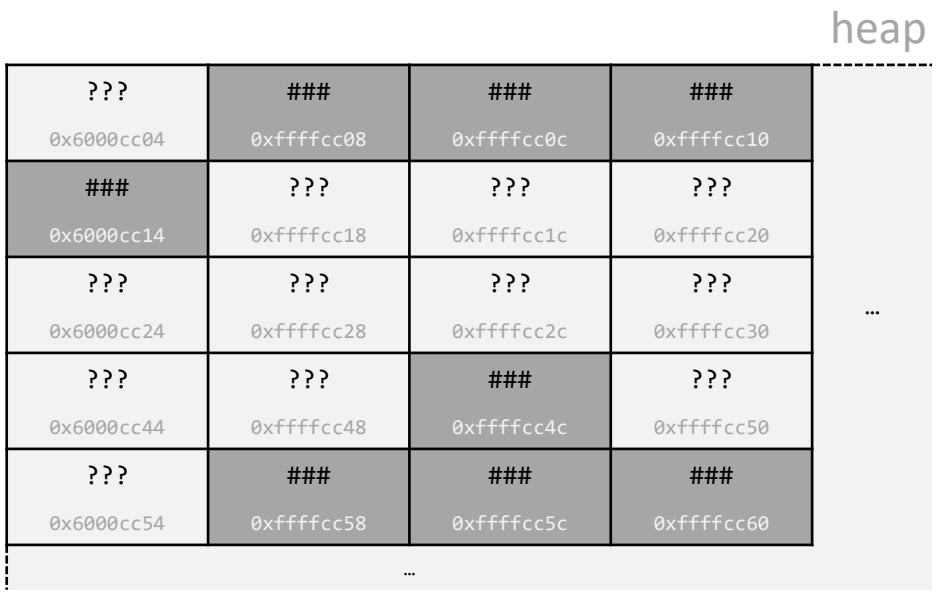
Alocar memória em funções

1/4

Problema

```c
int criarVetor(int *vetor) {
    int tamanho;
    scanf("%d", &tamanho);
    vetor = (int *) calloc(tamanho, sizeof (int));
    if (tamanho > 0 && vetor != NULL) {
        printf("\nt: %d p: %p", tamanho, vetor);
        return tamanho;
    }
    return 0;
}

int main() {
    int tamanho, *vetor = NULL;
    if ((tamanho = criarVetor(vetor)) > 0) {
        printf("\nt: %d p: %p", tamanho, vetor);
        free(vetor);
        vetor = NULL;
    } else {
        puts("vetor nao criado");
    }
    return 0;
}
```





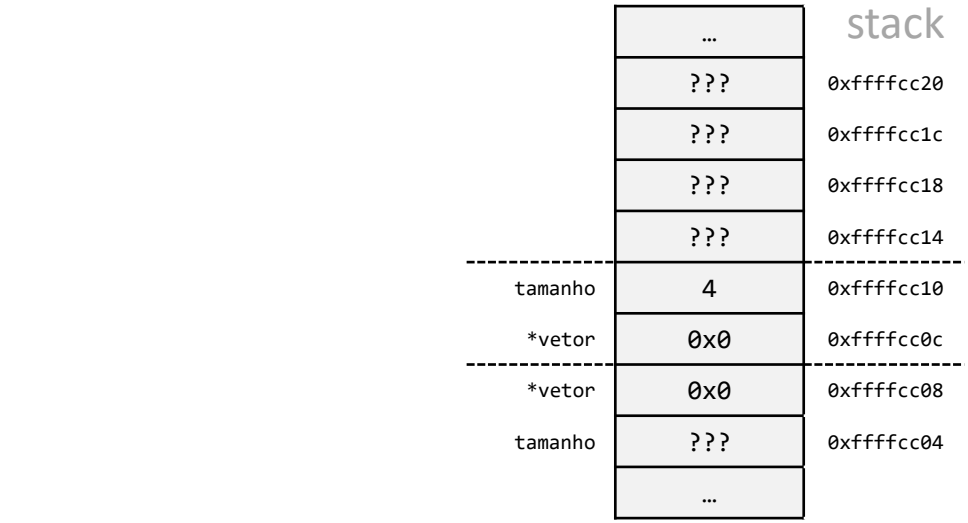
Alocar memória em funções

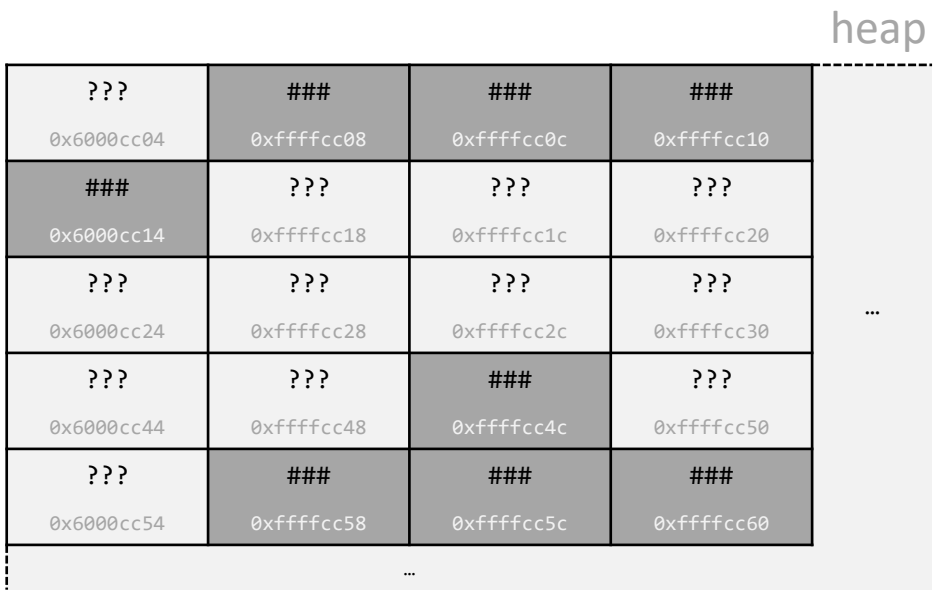
2/4

Problema

```c
int criarVetor(int *vetor) {
    int tamanho;
    scanf("%d", &tamanho);
    vetor = (int *) calloc(tamanho, sizeof (int));
    if (tamanho > 0 && vetor != NULL) {
        printf("\nt: %d p: %p", tamanho, vetor);
        return tamanho;
    }
    return 0;
}

int main() {
    int tamanho, *vetor = NULL;
    if ((tamanho = criarVetor(vetor)) > 0) {
        printf("\nt: %d p: %p", tamanho, vetor);
        free(vetor);
        vetor = NULL;
    } else {
        puts("vetor nao criado");
    }
    return 0;
}
```

```
4
```





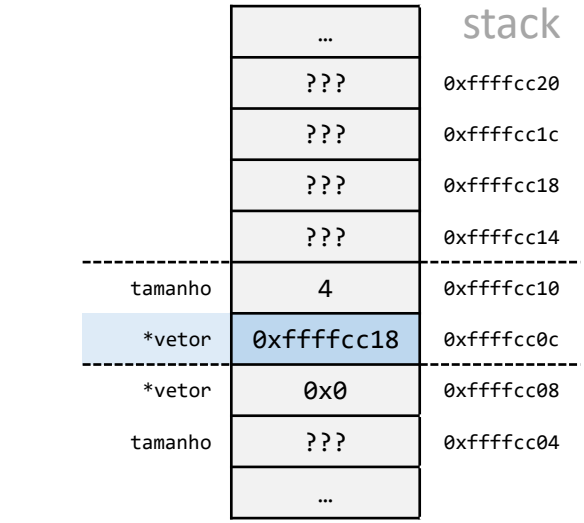
Alocar memória em funções

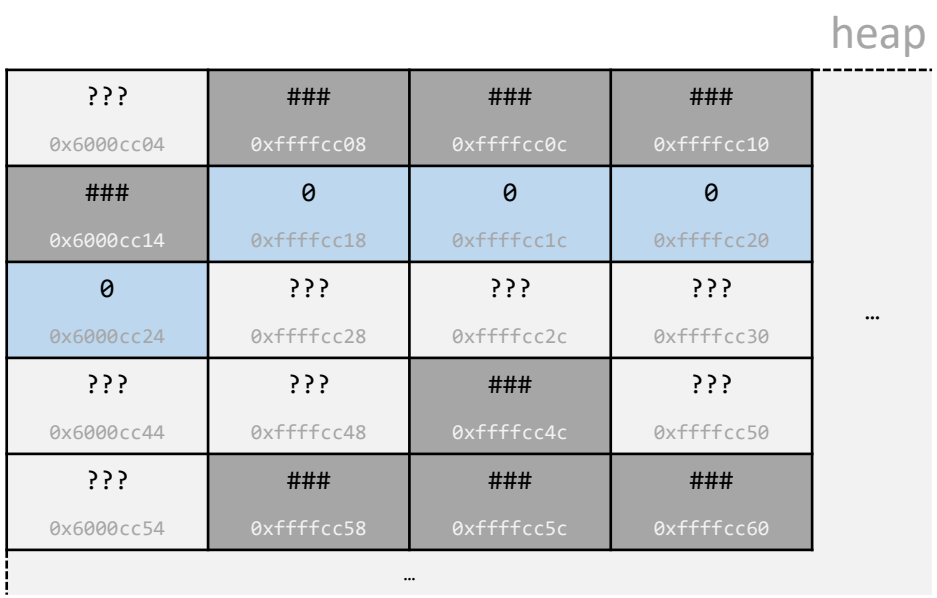
3/4

Problema

```c
int criarVetor(int *vetor) {
    int tamanho;
    scanf("%d", &tamanho);
    vetor = (int *) calloc(tamanho, sizeof (int));
    if (tamanho > 0 && vetor != NULL) {
        printf("\nt: %d p: %p", tamanho, vetor);
        return tamanho;
    }
    return 0;
}

int main() {
    int tamanho, *vetor = NULL;
    if ((tamanho = criarVetor(vetor)) > 0) {
        printf("\nt: %d p: %p", tamanho, vetor);
        free(vetor);
        vetor = NULL;
    } else {
        puts("vetor nao criado");
    }
    return 0;
}
```

```
4

t: 4 p: 0xffffcc18
```





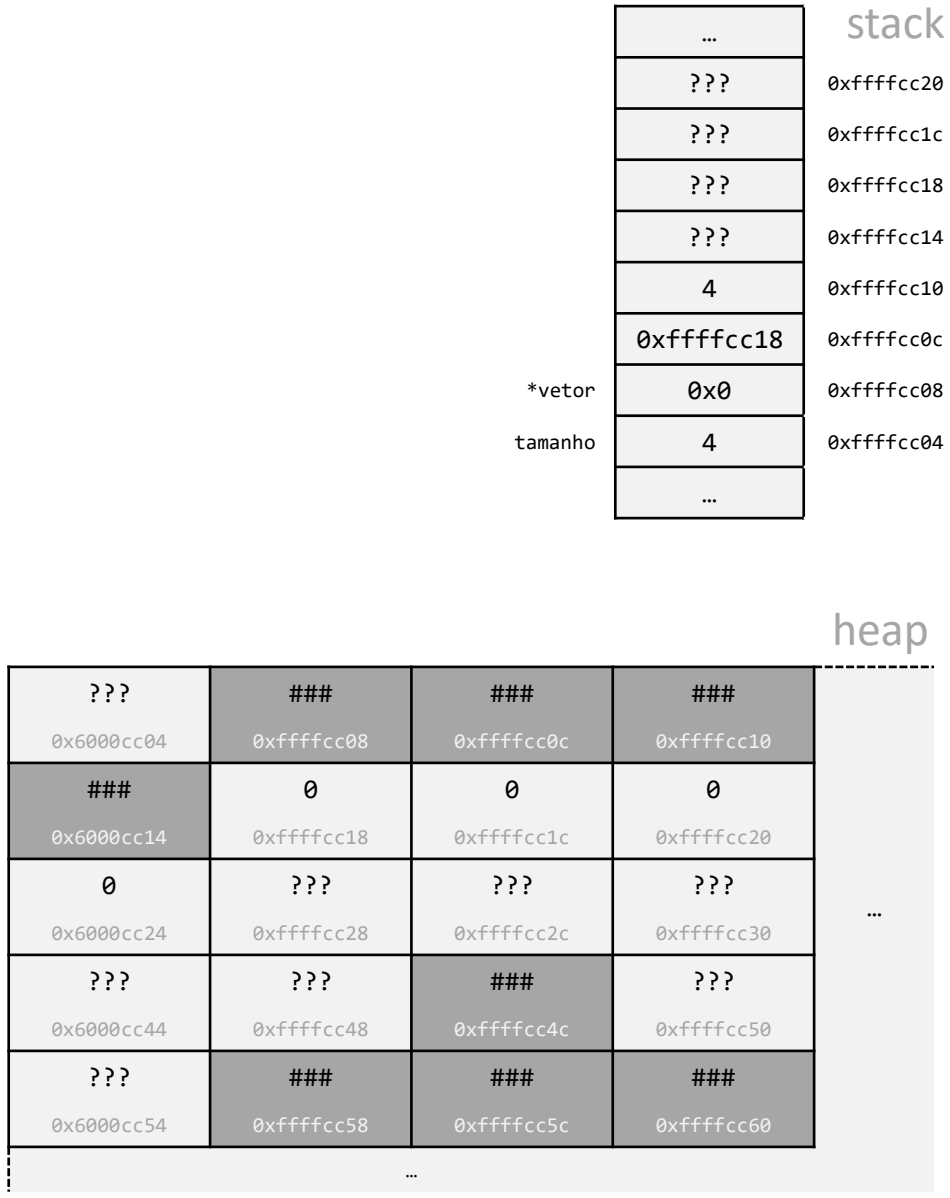
Alocar memória em funções

4/4

Problema

```c
int criarVetor(int *vetor) {
    int tamanho;
    scanf("%d", &tamanho);
    vetor = (int *) calloc(tamanho, sizeof (int));
    if (tamanho > 0 && vetor != NULL) {
        printf("\nt: %d p: %p", tamanho, vetor);
        return tamanho;
    }
    return 0;
}

int main() {
    int tamanho, *vetor = NULL;
    if ((tamanho = criarVetor(vetor)) > 0) {
        printf("\nt: %d p: %p", tamanho, vetor);
        free(vetor);
        vetor = NULL;
    } else {
        puts("vetor nao criado");
    }
    return 0;
}
```

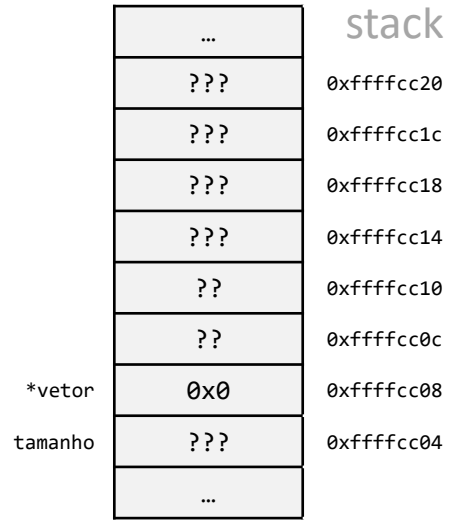
```
4

t: 4 p: 0xffffcc18
t: 4 p: 0x0
```

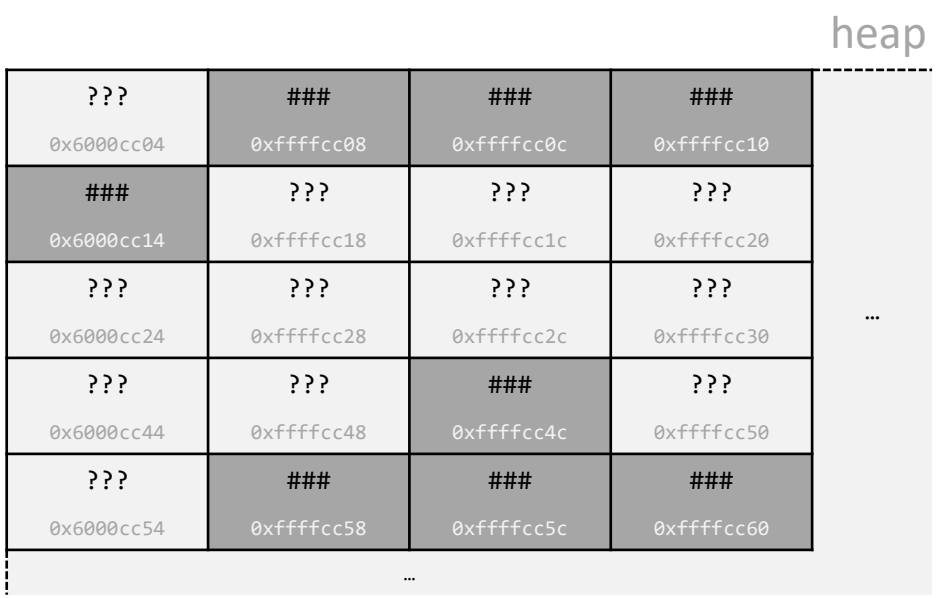


Alocar memória em funções

1/4

```c
int criarVetor(int **vetor) {
    int tamanho;
    scanf("%d", &tamanho);
    *vetor = (int *) calloc(tamanho, sizeof (int));
    if (tamanho > 0 && *vetor != NULL) {
        printf("\nt: %d p: %p", tamanho, *vetor);
        return tamanho;
    }
    return 0;
}

int main() {
    int tamanho, *vetor = NULL;
    if ((tamanho = criarVetor(&vetor)) > 0) {
        printf("\nt: %d p: %p", tamanho, vetor);
        free(vetor);
        vetor = NULL;
    } else {
        puts("vetor nao criado");
    }
    return 0;
}
```



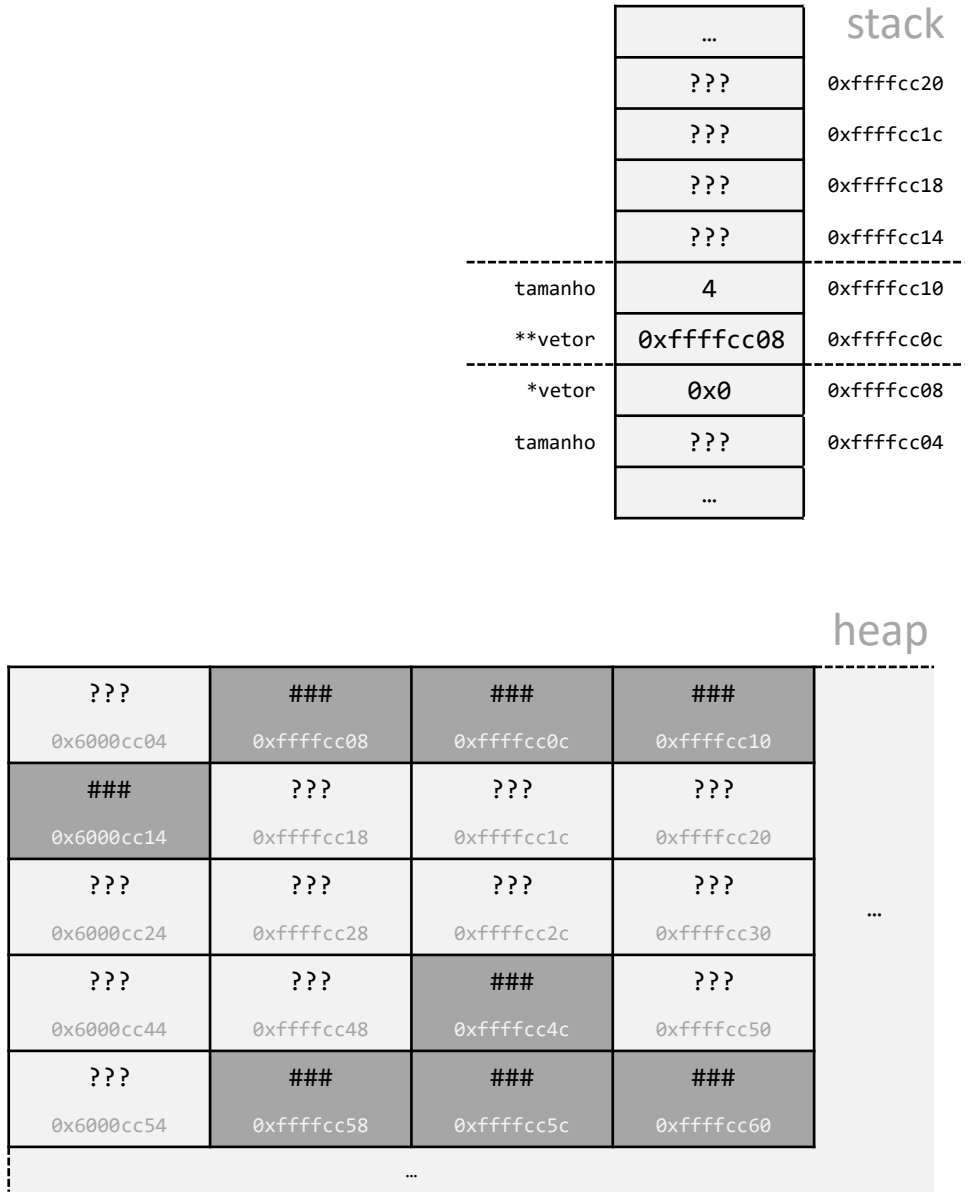


Alocar memória em funções

2/4

```c
int criarVetor(int **vetor) {
    int tamanho;
    scanf("%d", &tamanho);
    *vetor = (int *) calloc(tamanho, sizeof (int));
    if (tamanho > 0 && *vetor != NULL) {
        printf("\nt: %d p: %p", tamanho, *vetor);
        return tamanho;
    }
    return 0;
}

int main() {
    int tamanho, *vetor = NULL;
    if ((tamanho = criarVetor(&vetor)) > 0) {
        printf("\nt: %d p: %p", tamanho, vetor);
        free(vetor);
        vetor = NULL;
    } else {
        puts("vetor nao criado");
    }
    return 0;
}
```

```
4
```

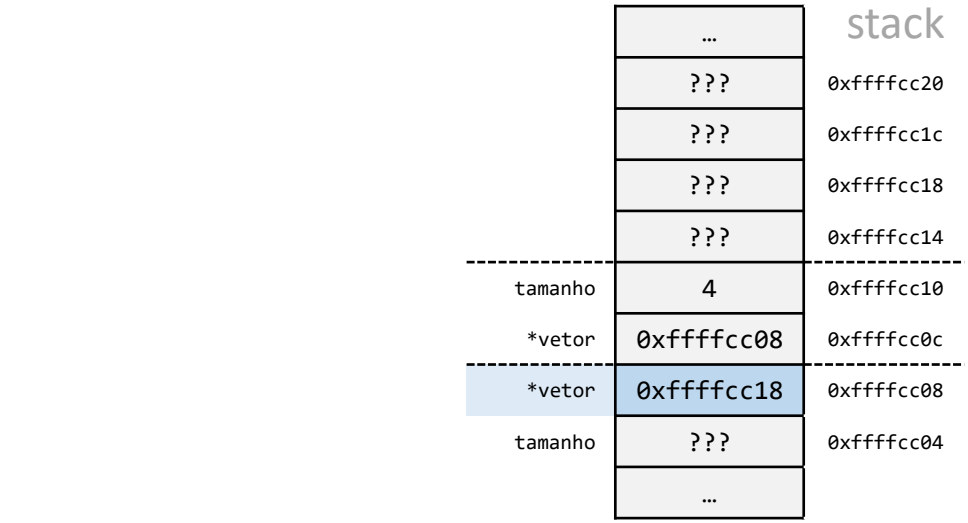


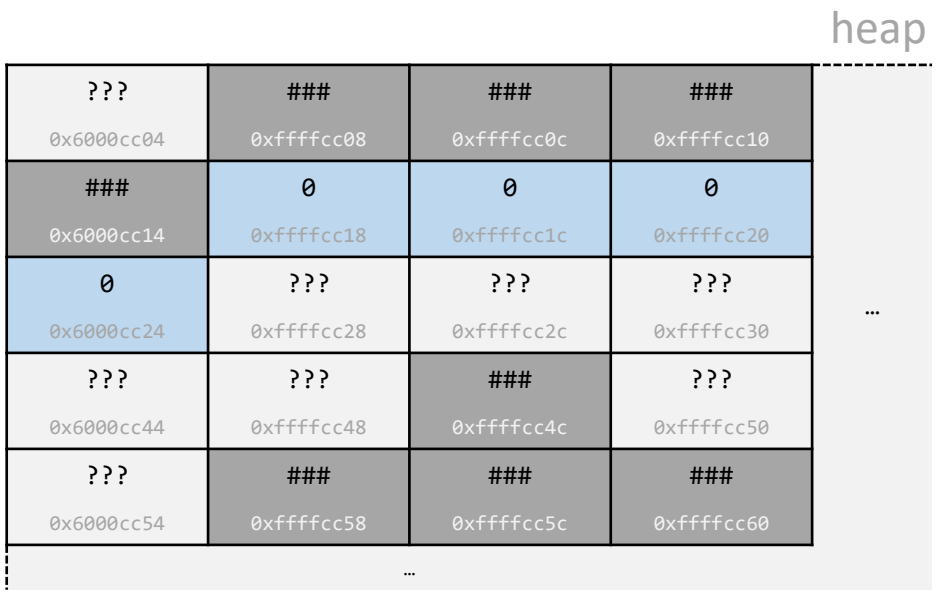
Alocar memória em funções

3/4

```c
int criarVetor(int **vetor) {
    int tamanho;
    scanf("%d", &tamanho);
    *vetor = (int *) calloc(tamanho, sizeof (int));
    if (tamanho > 0 && *vetor != NULL) {
        printf("\nt: %d p: %p", tamanho, *vetor);
        return tamanho;
    }
    return 0;
}

int main() {
    int tamanho, *vetor = NULL;
    if ((tamanho = criarVetor(&vetor)) > 0) {
        printf("\nt: %d p: %p", tamanho, vetor);
        free(vetor);
        vetor = NULL;
    } else {
        puts("vetor nao criado");
    }
    return 0;
}
```

```
4

t: 4 p: 0xffffcc18
```



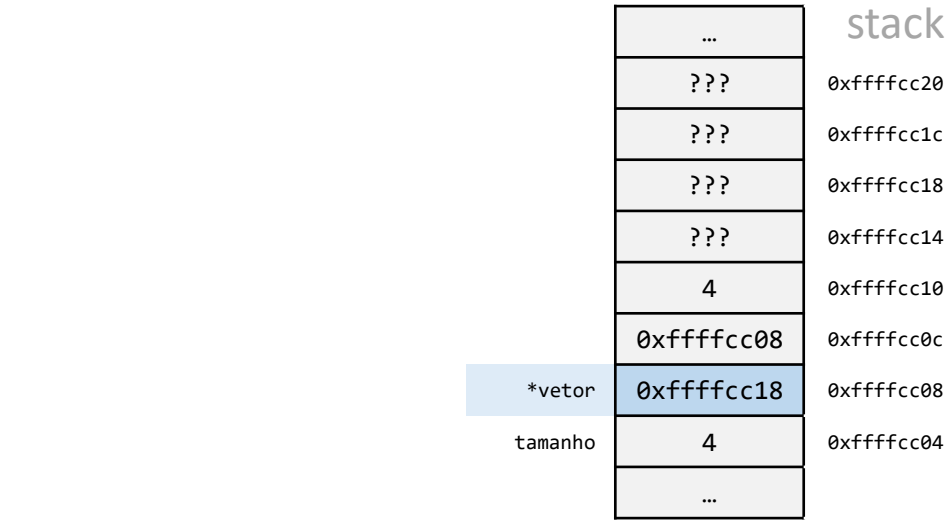


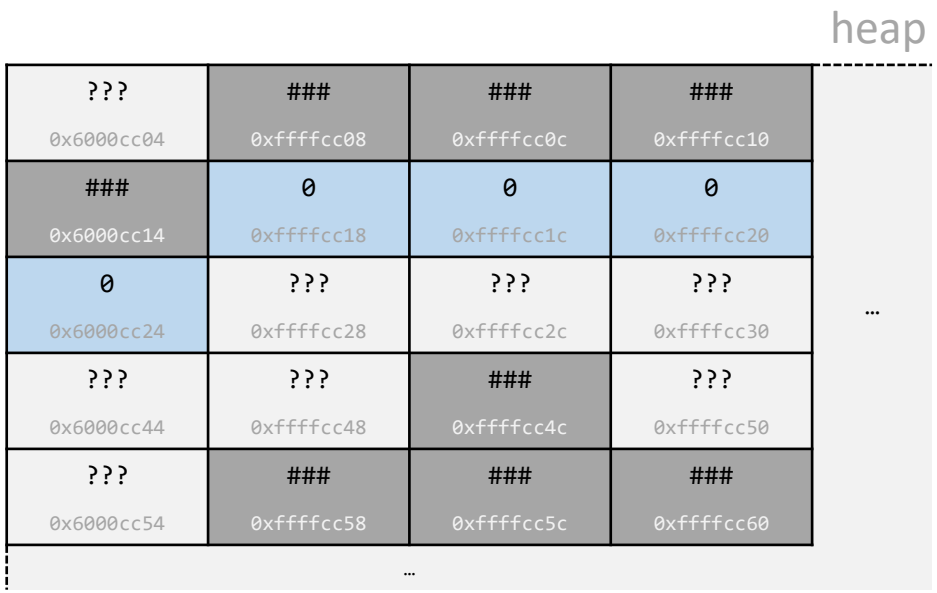
Alocar memória em funções

4/4

```c
int criarVetor(int **vetor) {
    int tamanho;
    scanf("%d", &tamanho);
    *vetor = (int *) calloc(tamanho, sizeof (int));
    if (tamanho > 0 && *vetor != NULL) {
        printf("\nt: %d p: %p", tamanho, *vetor);
        return tamanho;
    }
    return 0;
}
int main() {
    int tamanho, *vetor = NULL;
    if ((tamanho = criarVetor(&vetor)) > 0) {
        printf("\nt: %d p: %p", tamanho, vetor);
        free(vetor);
        vetor = NULL;
    } else {
        puts("vetor nao criado");
    }
    return 0;
}
```

```
4

t: 4 p: 0xffffcc18
t: 4 p: 0xffffcc18
```





Leitura recomendada

- (Capítulo 12) Damas, L. Linguagem C; FCA – Editora de Informática, Lda, 1999; ISBN 9789727221561.