Introdução à linguagem C (II)

Programação (L.EIC009)

Eduardo R. B. Marques, DCC/FCUP

Introdução à linguagem C (II)

- Arrays e strings
- Apontadores e arrays, aritmética de apontadores.
- Estruturas de dados com `struct`.
- Alocação de memória dinâmica.

Arrays e strings

Arrays em C

Um **"array"**, também chamado de variável indexada ou vector, é uma sequência de elementos contíguos em memória de tamanho fixo. Em C podemos declarar um array de tipo `int` e tamanho `10` por exemplo usando:

```c
int a[10]; // Valores arbitrários inicialmente
```

Variantes:

```c
// Com inicialização de todos os elementos.
int a[10] = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

// Mesmo array com tamanho de 10 implícito.
int a[] = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

// Com inicialização de alguns elementos apenas
// (restantes ficam a 0)
int a[10] = { 10, 20, 30 };
```

Acesso a arrays

Um array com `n` posições pode ser indexado usando o operador `[]` com índices de `0` a `n-1`.

Exemplo:

```
int a[3] = { 1, 2, 3 };
a[2] += a[1] + a[0]; // ficamos com a[2] = 6
```

A linguagem C **não valida** acessos a posições inválidos de um array. Quando tal acontece, o comportamento de um programa é indefinido. Podem ocorrer vários tipos de problemas comuns, por ex. erro de acesso a segmento inválido de memória ("segmentation fault"), acesso às posições de memória de outras variáveis do program, ou a corrupção da stack de execução.

```
int a[4];
a[4] = 1; // índice inválido
a[-1] = 0; // índice inválido
```

Arrays e funções

Um array é sempre passado por referência a uma função e não por valor. A função pode modificar o conteúdo do array passado como argumento.

```
void fill_with_zeros(int a[], int n) {
  for (int i = 0; i < n; i++)
     a[i] = 0;
}
int main(void) {
  int a[10];
  fill_with_zeros(a, 10);
  // a terá posições preenchidas com 0 depois da chamada
  ...
}
```

Precisamos frequentemente de passar um argumento extra a funções a identificar o tamanho de um array. Em C um array não tem associado um atributo para o seu tamanho (por ex. como em C# ou Java).

Arrays e funções (cont.)

Na prática, a passagem do array por referência equivale à passagem de um apontador para a primeira posição do array. A função anterior também pode ser expressa como:

```
void fill_with_zeros(int *a, int n) {
  for (int i = 0; i < n; i++) {
     a[i] = 0;
  }
}
```

O uso de `int a[]` em vez de `int *a` deixa mais claro que `a` se trata de um array. Ao usarmos `int *a` poderia ficar ambíguo se não estariamos a passar a referência (endereço) de uma variável simples de tipo `int` (que pode ser vista de qualquer forma como um array de tamanho 1!).

Exemplo de inversão de um array

(Ficha 2)

```
void invert(int n, const int a[], int b[]) {
   for (int i = 0; i < n; i++) {
     b[n - i - 1] = a[i];
   }
}
```

Nota: Ao chamarmos `invert(n, x, y)` devemos garantir que `x` e `y` são realmente arrays diferentes, tendo em conta a passagem por referência.

Uma chamada a `invert(n, x, x)` para um array `x` não funcionará correctamente dado que nesse caso `a` e `b` no corpo de `invert` irão-se referir ao mesmo array `x` e a inversão não será feita de forma correcta!

Exemplo de inversão de um array (cont.)

(também da Ficha 2)

Variante em que o array fonte é usado para a inversão:

```
void invert(int n, int a[]) {
   for (int i = 0; i < n / 2; i++) {
     int tmp = a[i];
     a[i] = a[n - i - 1];
     a[n - i - 1] = tmp;
   }
}
```

Strings

Em C **strings** são arrays de tipo `char` em que o valor `0`, também denotado por `'\0'`, é usado para indicar o fim da string.

Para um array de tipo `char` e tamanho `n`, o tamanho da string é dado por `l` (`l < n`) se `l` é primeira posição tal que `a[l] = 0`.

Quando usamos uma string constante, i.e. uma sequência de caracteres entre aspas, está implícita o valor `'\0'` no fim da sequência:

```
// Declarações equivalentes de strings
// com tamanho 3
char a[4] = { 'a', 'b', 'c', '\0' };
char b[4] = "abc";
```

Exemplo

Cálculo do tamanho de uma string (análogo a `strlen` na biblioteca de C) e cópia de strings (análogo a `strcpy`):

```
int length(const char str[]) {
  int pos = 0;
  while (str[pos] != '\0')
    pos++;
  }
  return pos;
}

void copy(char dst[], const char[] src) {
  int pos = 0;
  while (src[pos] != '\0') {
    dst[pos] = src[pos];
    pos++;
  }
}
```

Funções de manipulação de strings

A biblioteca standard de C tem várias funções para manipular strings como `strlen`, `strcpy`, `strcat`, `sprintf`, ...

O uso de muitas destas funções (ou de variantes como apresentamos no slide anterior) representa um risco de fiabilidade e segurança para um programa, porque não validam a possibilidade de "buffer overflows", i.e. a leitura ou escrita de posições em memória para além do limite do array.

Para programas "a sério" é aconselhável o uso de variantes destas funções que operam até um limite máximo de caracteres como `strncpy`, `strncat`, `snprintf` ...

Para saber mais veja por exemplo "Common vulnerabilities guide for C programmers - CERN Computer Security" ou a entrada na Wikipedia para "Buffer overflow.

Arrays multi-dimensionais

É possível declarar arrays com mais do que uma dimensão em C, por ex.:

```
int a[3][4];
int b[2][5] = {
  { 11, 12, 13, 14, 15},
  { 21, 22, 23, 24, 25},
  { 31, 32, 33, 34, 35}
};
```

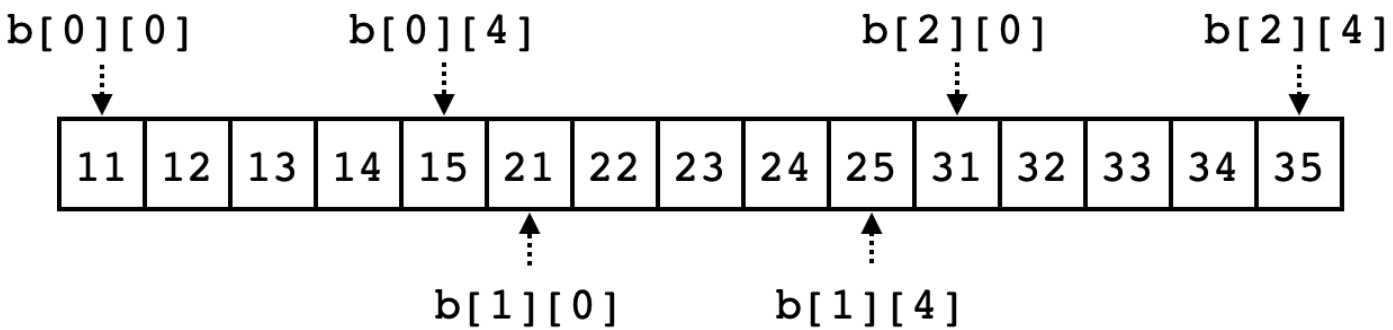
O esquema normal de indexação generaliza para várias dimensões, ex.

```
a[1][3] = b[0][4] + b[1][2] + b[2][0]; // 15 + 23 + 31
```

Arrays multi-dimensionais (cont.)

```
int b[2][5] = {
  { 11, 12, 13, 14, 15},
  { 21, 22, 23, 24, 25},
  { 31, 32, 33, 34, 35}
};
```

Um array multi-dimensional é representado em memória usando uma zona contígua de memória (como um array uni-dimensional):



Arrays multi-dimensionais (cont.)

Para passar um array multi-dimensional como argumento a uma função somos obrigados a passar o tamanho de todas as dimensões excepto eventualmente da primeira (de outra forma o compilador não consegue inferir a organização do array em memória).

Por exemplo, a seguinte definição é válida:

```
void fill_with_zeros(int m[][100], int rows) {
  for (int r = 0; r < rows; r++)
    for (int c = 0; c < 100; c++)
      m[r][c] = 0;
}
```

Arrays multi-dimensionais (cont.)

... mas esta outra não:

```
void fill_with_zeros(int m[][], int rows, int cols) {
  for (int r = 0; r < rows; r++)
    for (int c = 0; c < cols; c++)
      m[r][c] = 0;
}
```

pois resulta no seguintes erro de compilação:

```
error: array type has incomplete element type ‘int[]’
 d fill_with_zeros(int m[][], int rows, int cols) {
                           ^
y.c:1:26: note: declaration of ‘m’ as multidimensional array
must have bounds for all dimensions except the first
```

Estas limitações tornam o processamento de arrays multi-dimensionais pouco flexível. Para lidar com o problema, é possível representar arrays multi-dimensionais usando arrays de apontadores (ver mais à frente nestes slides).

Apontadores e arrays, aritmética de apontadores.

Revisão

Previamente, vimos como o uso de apontadores é útil para a passagem de argumentos a uma função, ex:

```
void min_max_sum
(int a, int b, int* min, int* max, int* sum) {
  *min = a < b ? a : b;
  *max = a > b ? a : b;
  *sum = a + b;
}

void some_func(void) {
  int maior, menor, s;
  min_max_sum(-123, 123, &menor, &maior, &s);
  ...
}
```

Apontadores e arrays

Podemos também usar apontadores para posições de arrays, e o operador `[]` pode indexar por sua vez apontadores (de forma análoga a arrays):

```
int a[3] = { 1, 3, 5};
int *p = &a[1];
*p = p[-1] + *p + p[0] + p[1];
```

Acima tomamos o endereço da posição `1` de `a` (`&a[1]`) para atribuição a `p`, e a seguir empregamos o operador de de-referenciação (`*`) e de indexação (`[]`) sobre `p`. A última linha acima é equivalente a modificar `a` da seguinte forma:

```
a[1] = a[0] + a[1] + a[1] + a[2];
```

Aritmética de apontadores

Podemos ainda empregar aritmética de apontadores.

Por exemplo o seguinte fragmento:

```
int a[3] = { 1, 3, 5};
int *p =  a; // mesmo que &a[0]
p++; // p agora aponta para a[1]
*p = 1;
p--; // p volta a apontar para a[0]
*p = 2;
p += 2; // p agora aponta para a[2]
*p = 3;
```

tem efeito equivalente ao de modificarmos `a` da seguinte forma:

```
a[1] = 1;
a[0] = 2;
a[2] = 3;
```

Aritmética de apontadores - exemplos (cont.)

```
int length(const char s[]) {
  const char* p = s; // mesmo que &s[0]
  while (*p != '\0') {
    p++;
  }
  return p - s; // mesmo que p - &s[0]
}
```

`p - s` dá-nos a diferença em número de items do tipo `char` entre `p` e `s` (`&s[0]`).

Aritmética de apontadores - exemplos (cont.)

```
void invert(int n, int[] a) {
  int* p = a;
  int* q = a + n - 1;
  while (p < q) {
    int tmp = *p;
    *p = *q;
    *q = tmp;
    p++;
    q--;
  }
}
```

O ciclo termina quando `q` apontar para uma zona de memória "atrás" de `p` ou igual a `p`.

Porque é que `p != q` em vez de `p < q` pode não funcionar? Pense no caso em que `n` tem valor par.

Aritmética de apontadores - exemplos (cont.)

```
void fill_with_zeros(int m[][100], int rows) {
  int* p = &m[0][0];
  int* q = p + rows * 100;
  while (p != q) {
    *p = 0;
    p++;
  }
}
```

Alternativa à implementação anterior: tiramos partido do facto de um array bi-dimensional ocupar uma zona contígua em memória.

Apontador nulo - “NULL pointer”

Muitas vezes é conveniente empregar o valor `0`, ou de forma equivalente a constante `NULL`, para denotar um apontador para “lado nenhum”.

O **apontador nulo (“null pointer”)** definido dessa forma denota muitas vezes a ausência de um resultado (ver próximo slide), ou o término de uma estrutura de dados com “apontadores ligados” (mais à frente nestes slides).

Apontador nulo - “NULL pointer” (cont.)

A seguinte função tem um comportamento análogo a `strchr` da biblioteca standard C - devolve um apontador para a primeira ocorrência de um caracter `c` em uma string `s` ou `NULL` caso o `c` não ocorra em `s`.

```
const char* find_char(const char s[], char c) {
  const char* p = s;
  while (*p != 0) {
    if (*p == c) {
      // character found - return p
      return p;
    }
    p++;
  }
  return NULL; // not found!
}
```

Apontador nulo - “NULL pointer” (cont.)

Ler ou escrever valores usando um apontador nulo leva frequentemente a um “segmentation fault” em tempo de execução, ex.:

```
int* p = NULL;
*p = 123; // ERRO
```

```
int *p = NULL;
int v = *p; // ERRO
```

No entanto, a semântica da linguagem C não dita qualquer comportamento esperado (é indefinida a este respeito).

Estruturas de dados usando `struct`

Definição

Podemos definir estruturas de dados com vários campos em C numa zona contígua de memória.

No seguinte exemplo definimos o tipo `struct coord2d` com dois campos `x` e `y` do tipo `double`.

```
struct coord2d {
  double x;
  double y;
};
```

Podemos declarar uma variável deste tipo:

```
struct coord2d a;
```

e aceder aos seus campos usando o operador `.`:

```
a.x = 1.5;
a.y = -2.5;
```

Definição (cont.)

Para cada tipo definido usando `struct` podemos evitar o uso repetido da palavra-chave `struct` em declarações empregando `typedef`.

Voltando ao exemplo anterior, podemos definir o tipo `coord2d` como sinónimo para `struct _coord2d`:

```
struct _coord2d {
  double x;
  double y;
};
typedef struct _coord coord2d;
...
coord2d c; // em vez de struct coord2d c
```

ou então definir `coord2d` como um tipo directamente:

```
typedef struct {
  double x;
  double y;
} coord2d;
```

Declaração com inicialização

Para:

```
typedef struct {
  double x;
  double y;
} coord2d;
```

podemos inicializar valores em declarações de `coord2d` caso queiramos:

```
// implicitamente valores para x e y
coord2d c = { 1.5, 2.5 };

// valores associados explicitamente
// a campos pelo seu nome
coord2d c = { .y = 2.5, .x = 1.5 };
```

Apontadores para tipos `struct`

De forma análoga a tipos primitivos de dados, podemos definir apontadores para tipos `struct`, ex.

```
coord2d c;
coord2d* p = &c;
(*p).x = 1.5;
(*p).y = 2.5;
```

Se `p` é um apontador para um tipo `struct` podemos usar o operador `->` para aceder a campos. De forma equivalente ao fragmento anterior, podemos escrever em alternativa:

```
coord2d c;
coord2d* p = &c;
p->x = 1.5;
p->y = 2.5;
```

Funções e tipos `struct`

Tipos `struct` podem ser passados por valor a funções e retornados também por valor, por ex.:

```
coord2d coord2d_add(coord2d a, coord2d b) {
  coord2d r;
  r.x = a.x + b.x;
  r.y = a.y + b.y;
  return r;
}
```

Tal não é no entanto de forma geral muito eficiente, dado que um tipo `struct` pode tomar muito espaço com passagem por valor.

Funções e tipos `struct` (cont.)

É muito mais usual e geralmente eficiente empregar apontadores, permitindo passagem por referência.

```
void coord2d_add
(coord2d* r, const coord2d* a, const coord2d* b) {
  r->x = a->x + b->x;
  r->y = a->y + b->y;
}
```

Exemplo - stack com capacidade fixa

(Ficha 2)

```
#define MAX_ELEMENTS 5
typedef struct {
  int elements[MAX_ELEMENTS]
  int size;
} stack;

void stack_init(stack* s) {
  s -> size = 0;
}
```

Exemplo - stack com capacidade fixa (cont.)

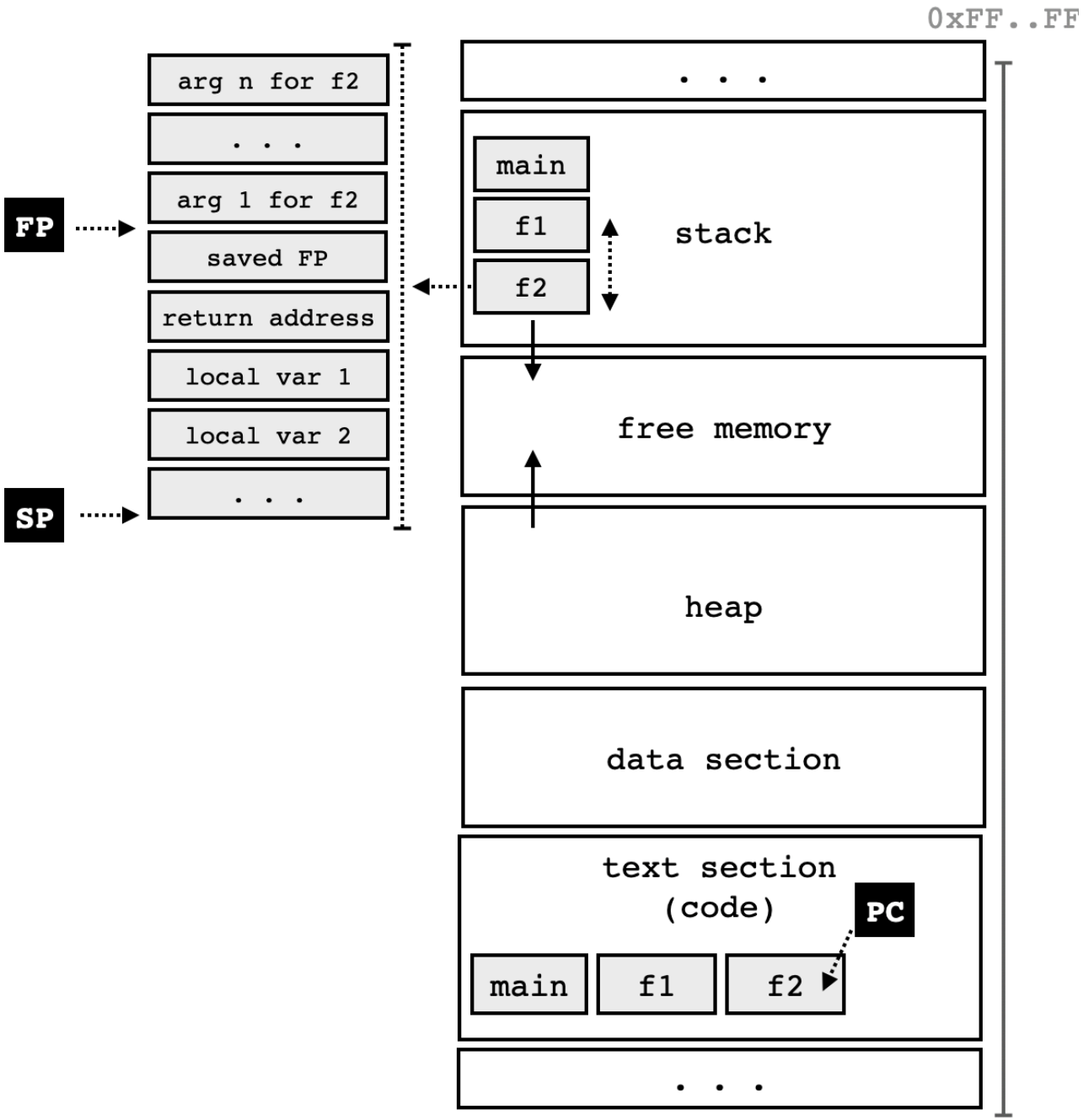
(Ficha 2)

```
bool stack_push(stack* s, int v) {
  if (s -> size == MAX_CAPACITY)
    return false; // stack is full
  (s -> elements)[s -> size] = v;
  (s -> size)++;
  return true;
}

bool stack_pop(stack* s, int* v) {
  if (s -> size == 0)
    return false; // stack is empty
  (s -> size)--;
  *v = (s -> elements)[s -> size];
  return true;
}
```

Alocação de memória dinâmica

Organização de memória de um programa



Organização de memória de um programa (cont.)

Secções de memória:

- **"Text"**: contém o código compilado para instruções máquina de um programa;
- **"Data"** : espaço usado por variáveis globais e constantes
- **"Stack"**: pilha de execução; contém "stack frames", uma por chamada a um procedimento de código na secção "Text".
  - Uma "stack frame" contém o espaço necessário à invocação de uma função, por ex. para variáveis locais de uma função, argumentos não passados em registos, ou o endereço de retorno da função.
  - Em programas com mais do que uma thread (não iremos falar disso) cada thread tem a sua stack.
- **"Heap"**: memória dinamicamente alocada; gerida automaticamente em linguagens com "garbage collection" ou de forma explícita pelo programador em linguagens como C e C++.

Gestão de memória dinâmica em C

Funções declaradas em `stdlib.h`:

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

void free(void *ptr);

void * malloc(size_t size);

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

`calloc`, `malloc`, `realloc`: alocam ou re-alocam dinamicamente um segmento de memória;

`free`: liberta memória dinamicamente alocada.

Alocação de memória com `malloc`

```
int n = ...;
int* p = malloc(n * sizeof(int));
for (int i = 0; i < n; i++)  p[i] = 0;
...
```

Uma chamada a `malloc(size)` aloca um segmento de memória com `size` bytes. Acima temos `size = n * sizeof(int)`, i.e., o espaço necessário a `n` items de tipo `int`.

O operador `sizeof` aplicado a um identificador de tipo `t` (como `int`) devolve o espaço necessário em bytes para representar um item de tipo `t`.

Alocação de memória com `calloc`

```
int n = ...;
int* p = calloc(n, sizeof(int));
...
```

Uma chamada a `calloc(count, size)` aloca um segmento de memória com `count` posições de tamanho `size`, i.e., tal como numa chamada a `malloc(count * size)`.

Ao contrário de `malloc`, `calloc` garante adicionalmente que todas as posições do segmento são inicialisada com o valor `0`; `malloc` ao invés retorna um segmento que pode ter valores arbitrários em memória.

Re-alocação de memória com `realloc`

```
int n = ...;
int* p = malloc(n * sizeof(int));
...
p = realloc(p,  2 * n * sizeof(int));
```

Uma chamada a `realloc(p, n)` devolve um novo segmento de memória de tamanho `n` caso o segmento de memória apontado por `p` tenha menos espaço que `n`, ou `p` caso contrário (`p` já terá espaço suficiente). No caso de re-alocação, a memória de `p` é copiada para o novo segmento e depois libertada.

Libertação de memória com `free`

```
int* p = ... ; // chamada a malloc, calloc, ou realloc
...            // Uso da memória
free(p);       // Libertação da memória
```

Um segmento de memória deve ser libertado com `free` quando já não é necessário na execução de um programa.

Problemas no uso de memória dinâmica

Conselho geral: "`free` after use, do not use after `free`."

"Free after use" - "memory leaks"

Um segmento de memória alocado deve ser libertado pelo programa quando não é mais necessário.

Quando tal não acontece, o segmento continuará a ocupar desnecessariamente memória. Temos uma "memory leak" ("fuga na memória").

"do not use after free" - "dangling references"

Após libertar um segmento de memória, não deveremos voltar a usá-lo.

Uma referência a memória já libertada é chamada uma "dangling reference" ("referência a balançar").

Exemplos a seguir

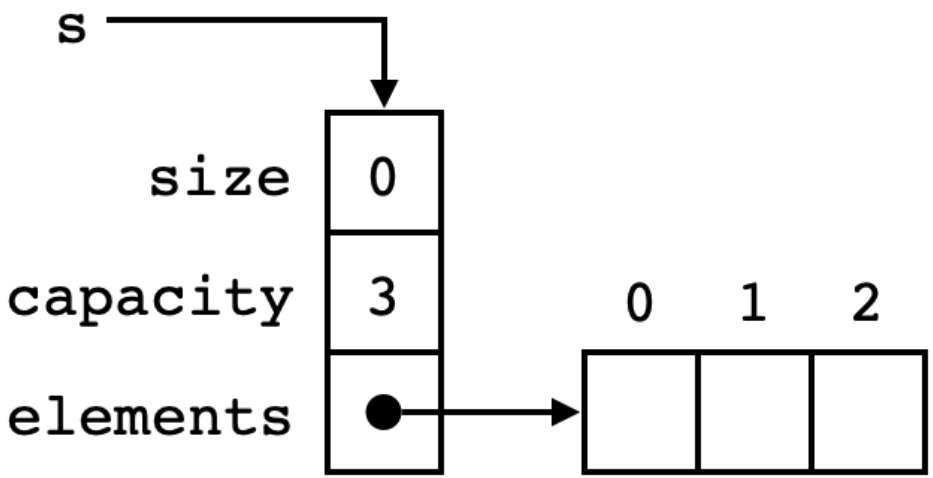
- Stacks sem limitação de capacidade
  - Usando um array que cresce dinamicamente
  - Usando uma lista ligada
- Implementação de uma matriz com tamanho dado em tempo de execução

Stack implementada com array dinâmico

```
#define INITIAL_CAPACITY 3
typedef struct {
  int size;
  int capacity;
  int* elements;
} stack;
...
stack* stack_create(void) {
  stack* s = malloc(sizeof(stack));
  s -> size = 0;
  s -> capacity = INITIAL_CAPACITY;
  s -> elements = malloc(s -> capacity * sizeof(int) );
  return s;
}
```

`stack_create` aloca espaço para a stack e depois para o seu array de elementos (`s -> elements`) com uma certa capacidade inicial (`INITIAL_CAPACITY` elementos de tipo `int`).

Stack implementada com array dinâmico (cont.)



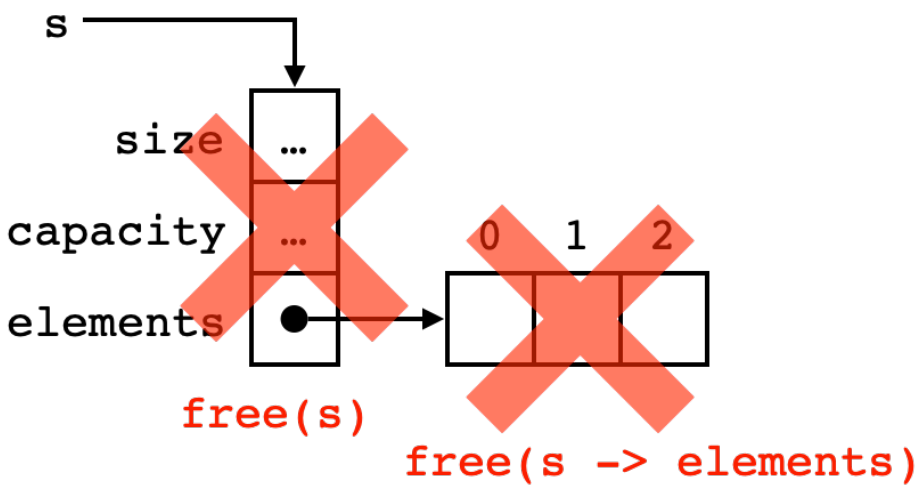

```
stack* stack_create(void) {
  stack* s = malloc(sizeof(stack));
  s -> size = 0;
  s -> capacity = INITIAL_CAPACITY;
  s -> elements = malloc(s -> capacity * sizeof(int) );
  return s;
}
```

Stack implementada com array dinâmico (cont.)

```
typedef struct {
  int size;
  int capacity;
  int* elements;
} stack;
...
void stack_destroy(stack* s) {
  free(s -> elements);
  free(s);
}
```

`stack_destroy` liberta o espaço ocupado pelo array (`s -> elements`) da stack e depois o segmento da própria stack (`s`).

Stack implementada com array dinâmico (cont.)




```
void stack_destroy(stack* s) {
  free(s -> elements);
  free(s);
}
```

Stack implementada com array dinâmico (cont.)

Se tivéssemos apenas a libertação de `s` em `stack_destroy`:

```
void stack_destroy(stack* s) {
  // free(s->elements); MEMORY LEAK
  free(s);
}
```

o espaço apontado por `s->elements` passaria a ser uma "memory leak".

Em alternativa, se a ordem das chamadas a `free` fosse invertida:

```
void stack_destroy(stack* s) {
  free(s);
  free(s->elements); // DANGLING REFERENCE
}
```

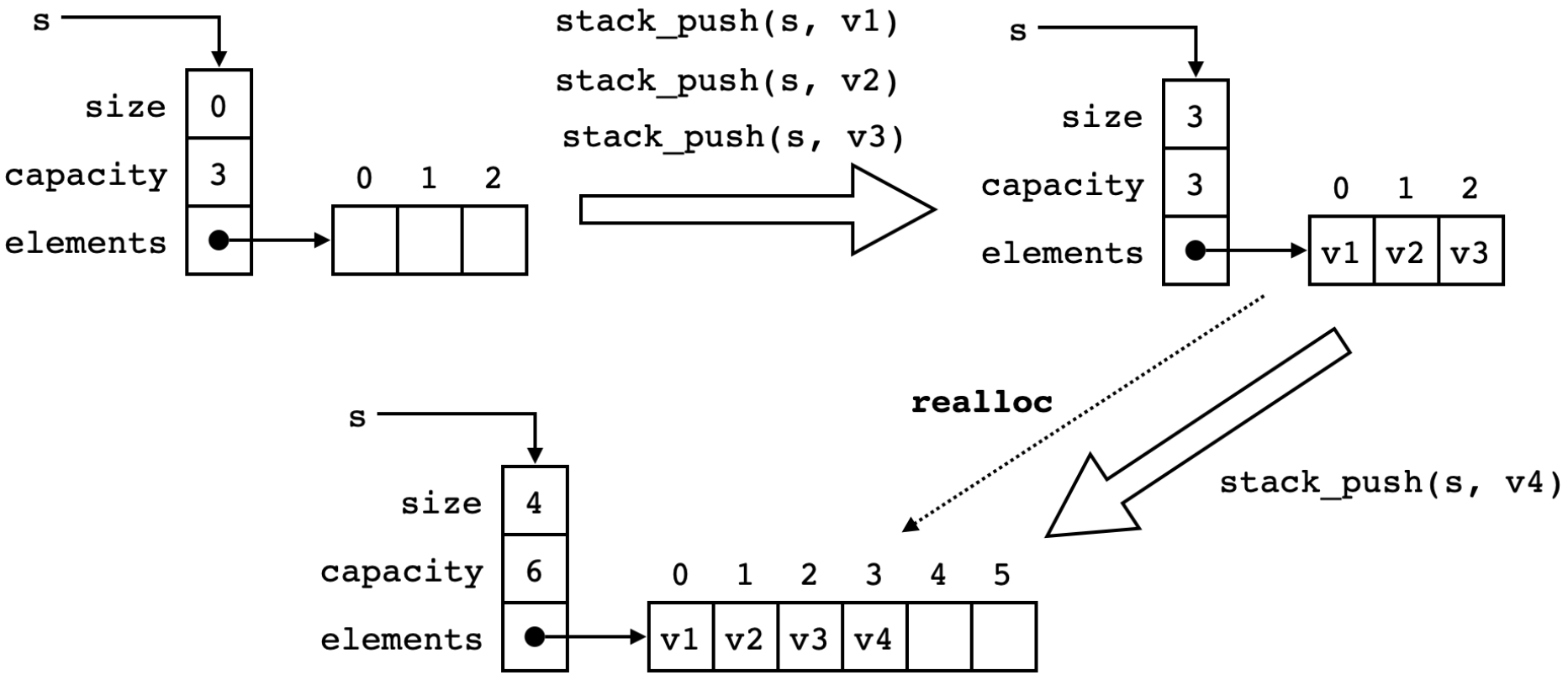
`s` seria uma "dangling reference" na 2ª chamada a `free`.

Stack implementada com array dinâmico (cont.)

```
void stack_push(stack* s, int v) {
  if (s -> size == s -> capacity) {
    s -> capacity = 2 * s -> capacity;
    s -> elements = realloc(s -> elements,
                            s -> capacity * sizeof(int));
  }
  s -> elements[s -> size] = v;
  s -> size++;
}
```

Quando necessário, `stack_push` emprega `realloc` para fazer crescer o array usado pela stack. No código o array cresce para o dobro (uma estratégia comum para evitar re-alocações sucessivas).

Stack implementada com array dinâmico (cont.)




```
if (s -> size == s -> capacity) {
  s -> capacity = 2 * s -> capacity;
  s -> elements = realloc(s -> elements,
                          s -> capacity * sizeof(int));
}
```

Stack implementada com lista ligada

```
struct _stacknode {
  int value;
  struct _stacknode* next;
};
typedef struct _stacknode stacknode;
typedef struct {
  int size;
  stacknode* top;
} stack;
```

A stack emprega uma lista simplesmente ligada de nós para guardar os elementos. O campos `size` indica o número de elements (nós usados) e `top` referencia o primeiro nó.

Stack implementada com lista ligada (cont.)

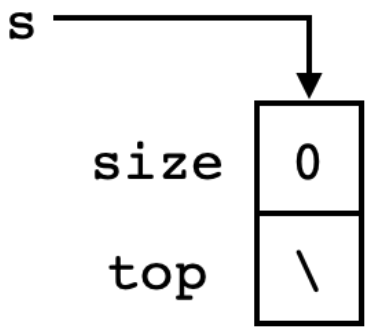
Podemos ter a seguinte função para alocar e inicializar uma stack:

```
stack* stack_create(void) {
  stack* s = malloc(sizeof(stack));
  s->size = 0;
  s->top = NULL;
}
```

Como `calloc` garante o preenchimento de memória com `0` (a propósito relembre-se que `NULL` equivale a `0`), podemos ter alternativamente:

```
stack* stack_create(void) {
  return calloc(1, sizeof(stack));
}
```

Stack implementada com lista ligada (cont.)



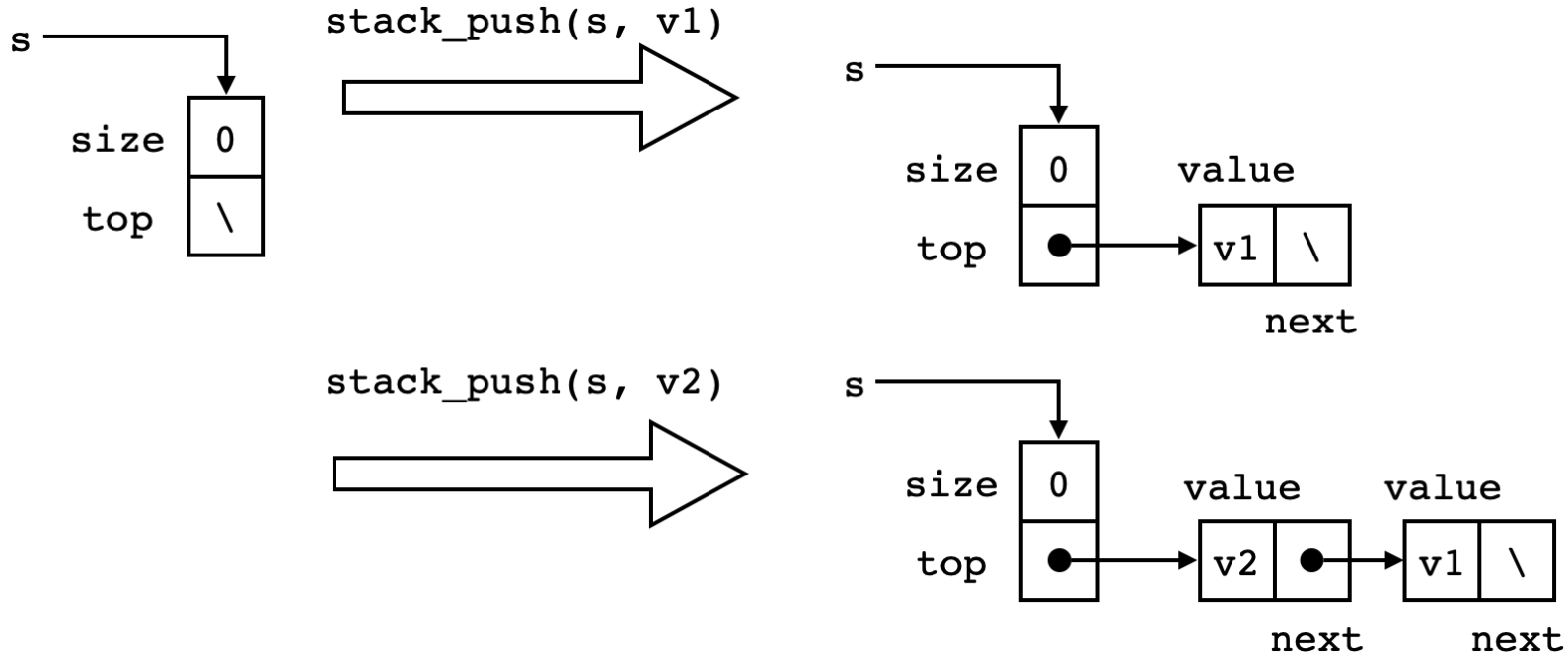

```
stack* stack_create(void) {
  stack* s = malloc(sizeof(stack));
  s->size = 0;
  s->top = NULL;
}
```

Stack implementada com lista ligada (cont.)

```
void stack_push(stack* s, int v) {
  stacknode* new_node = malloc(sizeof(stacknode));
  new_node -> value = v;
  new_node -> next = s -> top;
  s -> top = new_node;
  s -> size++;
}
```

`stack_push`: aloca espaço novo nó quando elemento é adicionada à stack.

Stack implementada com lista ligada (cont.)




```
void stack_push(stack* s, int v) {
  stacknode* new_node = malloc(sizeof(stacknode));
  new_node -> value = v;
  new_node -> next = s -> top;
  s -> top = new_node;
  s -> size++;
}
```

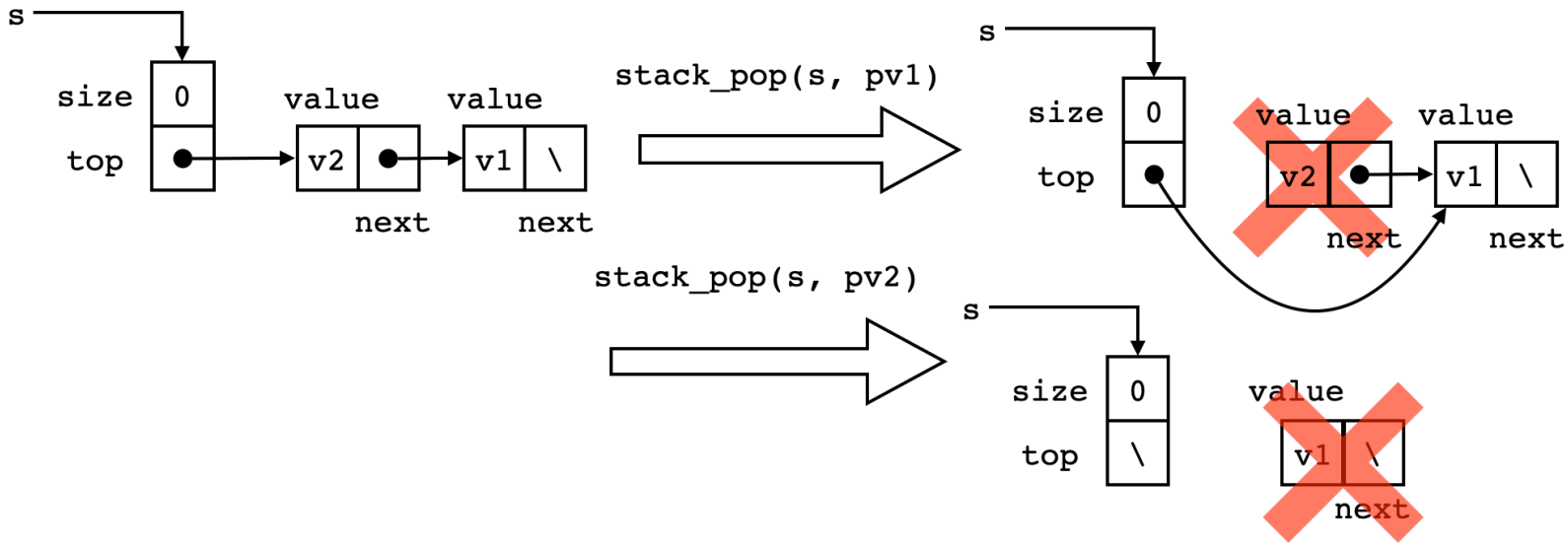
Stack implementada com lista ligada (cont.)

```
bool stack_pop(stack* s, int* pv) {
  if (s -> size == 0) {
    return false;´
  }
  stacknode* old_top = s -> top;
  *pv = old_top -> value;
  s -> top = old_top -> next;
  s -> size--;
  free(old_top);
  return true;
}
```

`stack_pop` liberta nó quando é removido um elemento.

A chamada `free(old_top)` deve acontecer (para evitar uma "memory leak") e só quando realmente a memória em causa não corre o risco de ser mais referenciada (para evitar uma "dangling reference").

Stack implementada com lista ligada (cont.)



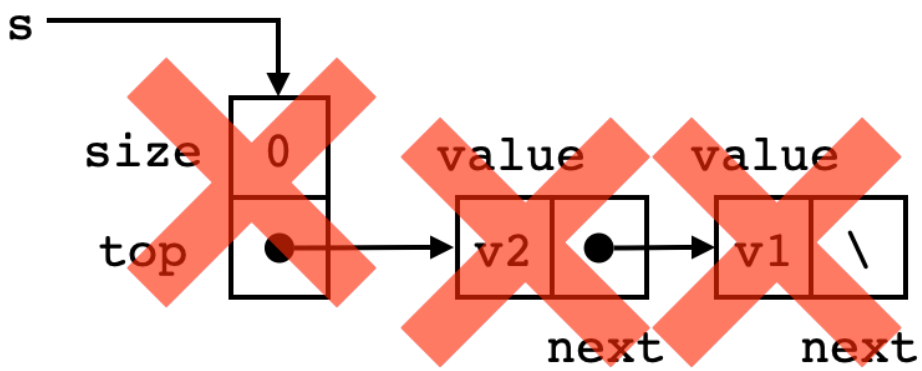

```
bool stack_pop(stack* s, int* pv) {
  ...
  stacknode* old_top = s -> top;
  *pv = old_top -> value;
  s -> top = old_top -> next;
  s -> size--;
  free(old_top);
  return true;
}
```

Stack implementada com lista ligada (cont.)

```
typedef struct {
  int size;
  stacknode* top;
} stack;
...
void stack_destroy(stack* s) {
  stacknode* n = s -> top;
  while (n != NULL) {
    stacknode* aux = n -> next;
    free(n);
    n = aux;
  }
  free(s);
}
```

`stack_destroy` liberta a memória associada a uma stack: todos os nós que estejam a ser usados e só depois o segmento da `stack` propriamente dito.

Stack implementada com lista ligada (cont.)



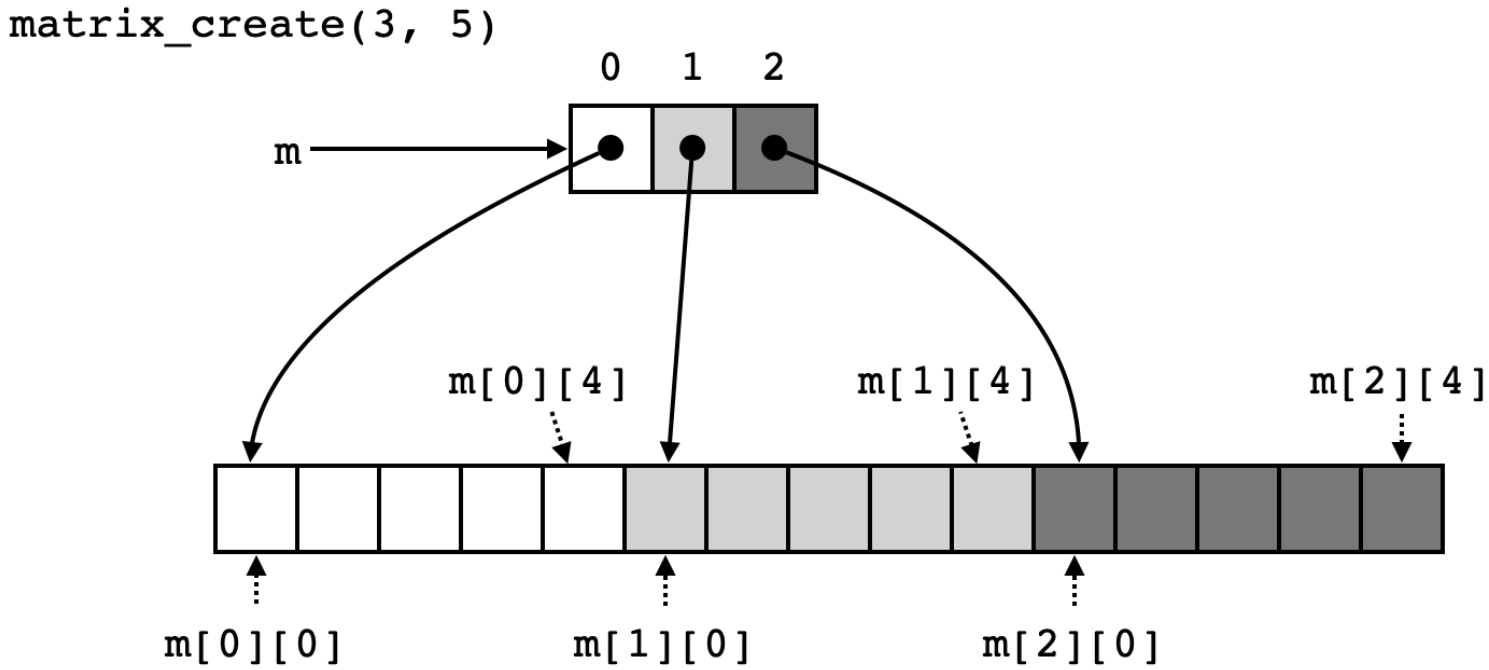

```
void stack_destroy(stack* s) {
  stacknode* n = s -> top;
  while (n != NULL) {
    stacknode* aux = n -> next;
    free(n);
    n = aux;
  }
  free(s);
}
```

Implementação de uma matriz

Alocação de uma matriz de tipo `double` e dimensão `lines` $\times$ `cols`.

```
double** matrix_create(int rows, int cols) {
  double** m = malloc(rows * sizeof(double*));
  double* values = malloc( rows * cols * sizeof(double));
  for (int r = 0; r < rows; r++) {
    m[r] = &values[r * cols];
  }
  return m;
}
```

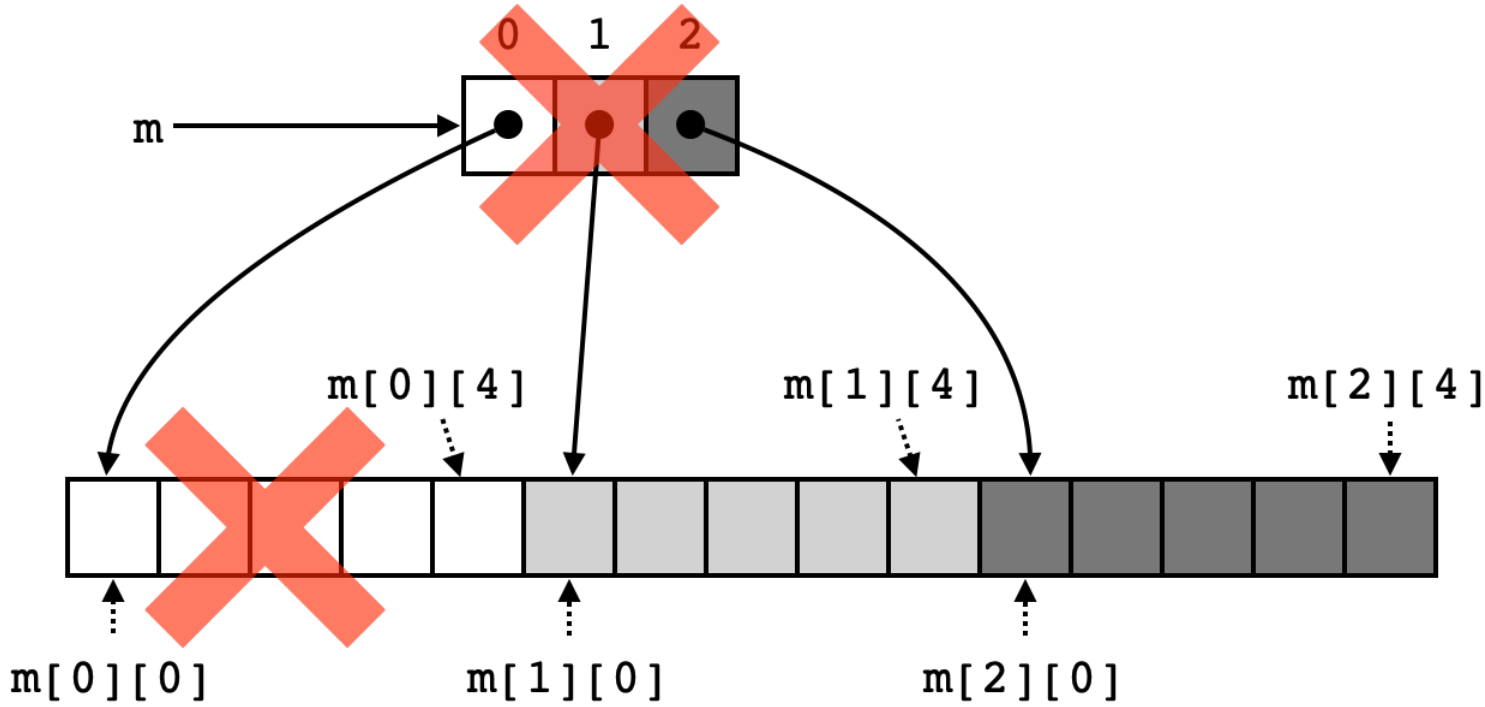
Implementação de uma matriz (cont.)



Aloca-se um array de apontadores `m` de tipo `double**` e depois um array de valores `v`.

Cada linha da matriz `l` é dada pelo apontador `m[l]` que referencia uma posição apropriada em `values`.

Implementação de uma matriz (cont.)




```
void matrix_destroy(double** m) {
  free(m[0]);
  free(m);
}
```

`m[0]` corresponde ao array de valores para a matriz.

`m` corresponde ao array de apontadores.