Melhores momentos

AULA 4

Conceitos discutidos

- mais recursão: mdc e algoritmo de Euclides
- um pouco de análise (experimental) de algoritmos
- structs
- argumentos na linha de comando

Registros e structs

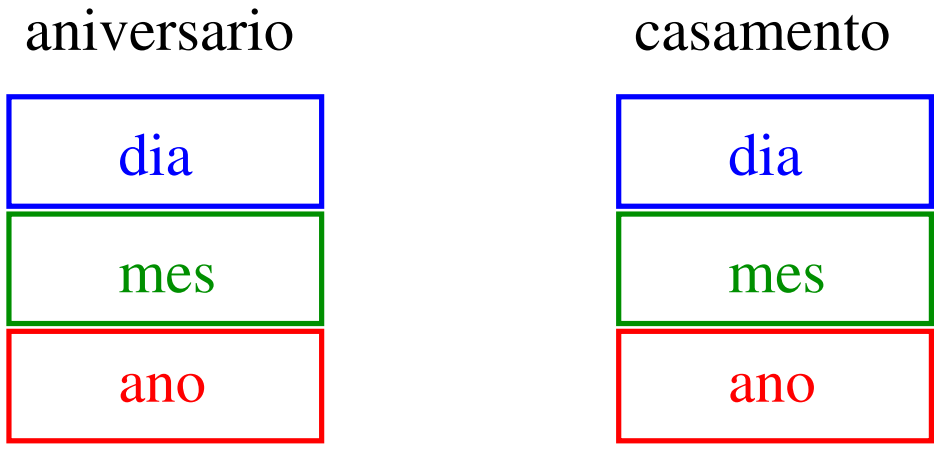
Um **registro** (= *record*) ou **structs** é uma coleção de diversas variáveis, possivelmente de tipos diferentes.

```
struct data {
   int dia;
   int mes;
   int ano;
};

typedef struct data Data;

Data aniversario;
Data casamento;
```

Estruturas e typedef

Um modo ainda mais compacto de fazer isso:

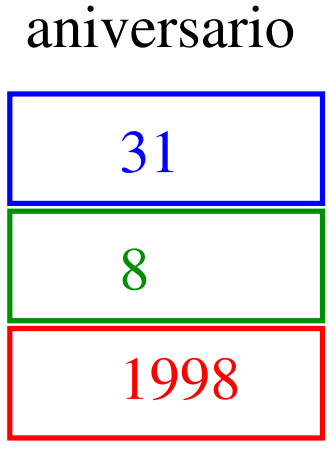
```
typedef struct {
    int dia, mes, ano;
} Data;

Data aniversario;
Data casamento;
```



Campos de uma estrutura

É fácil atribuirmos valores aos campos de uma estrutura:

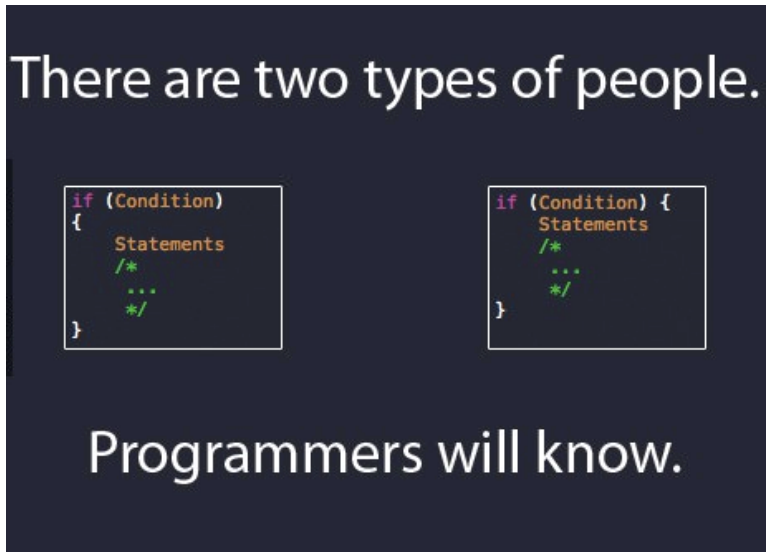
```
aniversario.dia = 31;
aniversario.mes = 8;
aniversario.ano = 1998;
```



AULA 5

Hoje

- endereços e ponteiros
- alocação dinâmica de memória



Fonte: http://www.geek-jokes.com/

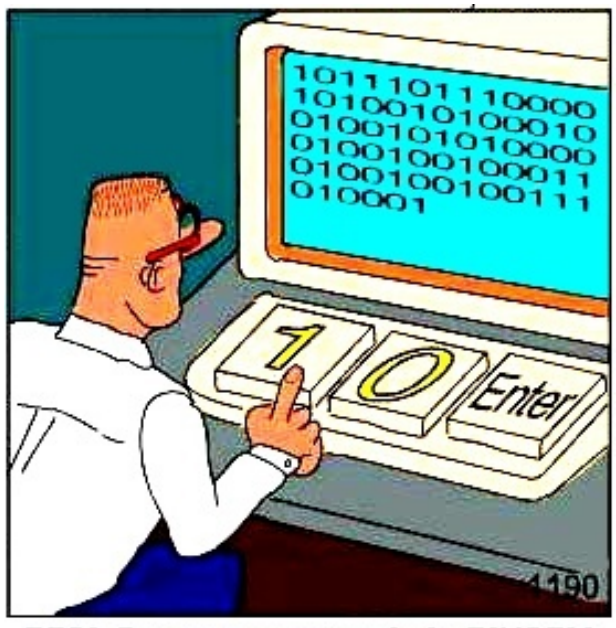
Endereços e Ponteiros

PF Apêndice D

http://www.ime.usp.br/~pf/algoritmos/aulas/pont.html

The C programming Language
Brian W. Kernighan e Dennis M. Ritchie
Prentice-Hall

Endereços



REAL Programmers code in BINARY.

Fonte: http://www.pinterest.com/iqnection/

Endereços

A memória de qualquer computador é uma sequência de bytes. Os bytes são numerados sequencialmente.

Endereços

A memória de qualquer computador é uma sequência de bytes. Os bytes são numerados sequencialmente.

O número de um byte é o seu **endereço**.

Endereços

A memória de qualquer computador é uma sequência de bytes. Os bytes são numerados sequencialmente.

O número de um byte é o seu **endereço**.

Cada objeto na memória do computador ocupa um certo número de bytes consecutivos.

```
printf("sizeof(char)   = %d", sizeof(char));
printf("sizeof(int)    = %d", sizeof(int));
printf("sizeof(float)  = %d", sizeof(float));
printf("sizeof(double) = %d", sizeof(double));
printf("sizeof(char *) = %d", sizeof(char));
printf("sizeof(int *)  = %d", sizeof(int));
```

Endereços

A memória de qualquer computador é uma sequência de bytes. Os bytes são numerados sequencialmente.

O número de um byte é o seu **endereço**.

Cada objeto na memória do computador ocupa um certo número de bytes consecutivos.

```
sizeof(char)   = 1
sizeof(int)    = 4
sizeof(float)  = 4
sizeof(double) = 8
sizeof(char *) = 4
sizeof(int *)  = 4
```

Endereços

Cada objeto na memória do computador tem um **endereço**.

Por exemplo, depois das declarações,

```
char c;
int i;

struct {
   int x, y;
} ponto;

int v[3];
```

Endereços

Cada objeto na memória do computador tem um **endereço**.

Por exemplo, depois das declarações,

```
char c;
int i;

struct {
   int x, y;
} ponto;

int v[3];
```

os endereços das variáveis poderiam ser:

```
end. c       = 0xbffd499f
end. i       = 0xbffd4998
end. ponto   = 0xbffd4990
end. ponto.x = 0xbffd4990
end. ponto.y = 0xbffd4994
end. v[0]    = 0xbffd4980
end. v[1]    = 0xbffd4984
end. v[2]    = 0xbffd4988
```

Endereço de uma variável

O endereço de uma variável é dado pelo operador `&`.

Se `i` é uma variável então `&i` é o seu endereço.

Endereço de uma variável

O endereço de uma variável é dado pelo operador `&`.

Se `i` é uma variável então `&i` é o seu endereço.

No exemplo anterior,

`&i` vale `0xbffd4998`

`&ponto` vale `0xbffd4990`

`&ponto.x` vale `0xbffd4990`

`&v[0]` vale `0xbffd4980`

```
end. c       = 0xbffd499f
end. i       = 0xbffd4998
end. ponto   = 0xbffd4990
end. ponto.x = 0xbffd4990
end. ponto.y = 0xbffd4994
end. v[0]    = 0xbffd4980
end. v[1]    = 0xbffd4984
end. v[2]    = 0xbffd4988
```

scanf

O segundo argumento da função de biblioteca `scanf` é o endereço da posição na memória onde devem ser depositados os objetos lidos no dispositivo padrão de entrada:

```
int i;

scanf("%d", &i);
printf("end. i=%p cont. i=%d",
                (void *)&i, i);
```

`%p` = imprime endereço em hexadecimal

Ponteiros

Fonte: http://xkcd.com/138/

Ponteiros

Um **ponteiro** (= apontador = *pointer*) é um tipo especial de variável que armazena endereços.

Ponteiros

Um **ponteiro** (= apontador = *pointer*) é um tipo especial de variável que armazena endereços.

Um ponteiro pode ter o valor especial

`NULL`

que não é o endereço de lugar algum.

Ponteiros

Um **ponteiro** (= apontador = *pointer*) é um tipo especial de variável que armazena endereços.

Um ponteiro pode ter o valor especial

`NULL`

que não é o endereço de lugar algum.

A constante `NULL` está definida no arquivo-interface `stdlib` e seu valor é 0 na maioria dos computadores.

Ponteiros

Se um ponteiro `p` armazena o endereço de uma variável `i`, podemos dizer "`p` *aponta para* `i`" ou "`p` *é o endereço de* `i`".

Ponteiros

Se um ponteiro `p` tem valor diferente de `NULL`, então

`*p`

é o objeto apontado por `p`.

Ponteiros

Há vários tipos de ponteiros: para caracteres, para inteiros, para ponteiros para inteiros, para registros, etc.

Ponteiros

Há vários tipos de ponteiros: para caracteres, para inteiros, para ponteiros para inteiros, para registros, etc.

Para declarar um ponteiro `p` para um inteiro, escrevemos

```
int *p;
```

Ponteiros

Há vários tipos de ponteiros: para caracteres, para inteiros, para ponteiros para inteiros, para registros, etc.

Para declarar um ponteiro `p` para um inteiro, escrevemos

```
int *p;
```

Para declarar um ponteiro `p` para uma estrutura `ponto`, escrevemos

```
struct ponto *p;
```

Exemplos

Eis um jeito bobo de fazer "c = a+b":

```
int *p; /* p eh ponteiro para um int */
int *q;
p = &a; /* conteudo p == endereco de a */
q = &b; /* q aponta para b */
c = *p + *q;
```

Exemplos

Outro exemplo bobo:

```
int *p;

int **r; /* r eh um ponteiro para um
            ponteiro para um inteiro */

p = &a;  /* p aponta para a */

r = &p;  /* r aponta para p e
            *r aponta para a */

c = **r + b;
```

Troca errada

```
void troca (int i, int j) { /*  errado!  */
   int temp;

   temp = i;
   i = j;
   j = temp;
}
```

Troca errada

```
void troca (int i, int j) { /*  errado!  */
   int temp;

   temp = i;
   i = j;
   j = temp;
}
```

Chamada da função:

```
    a = 10; b = 20;
    troca(a, b);
```

mas não tem efeito nenhum...

Troca certa

```
void troca (int *i, int *j) { /*  certo!  */
   int temp;

   temp = *i;
   *i = *j;
   *j = temp;
}
```

Troca certa

```
void troca (int *i, int *j) { /*  certo!  */
   int temp;

   temp = *i;
   *i = *j;
   *j = temp;
}
```

Chamada da função:

```
    a = 10; b = 20;
    troca(&a, &b);
```

Vetores e endereços

Em C, existe uma relação muuuito grande
entre ponteiros e vetores.

A declaração

```
int v[10];
```

define um bloco de 10 objetos consecutivos na memória
de nomes

`v[0]`, `v[1]`,..., `v[9]`

Vetores e endereços

Suponha que `p` é um ponteiro para um inteiro

```
int *p;
```

Então a atribuição

```
p = &v[0];
```

faz com que `p` contenha o endereço de `v[0]`.

Aritmética de ponteiros

Se `p` aponta para um elemento do vetor, então
`p+1` aponta para o elemento seguinte,
`p+i` aponta para o `i`-ésimo elemento depois de `p`,
`p-i` para o `i`-ésimo elemento antes de `p`.

Assim, `*(p+1)` é `v[1]`, `*(p+2)` é `v[2]`, ...

Aritmética de ponteiros

O significado de "somar 1 a um ponteiro" é que `p+1` aponta para o próximo objeto, independente do número de bytes do objeto.

Assim, `*(p+1)` é `v[1]`, `*(p+2)` é `v[2]`, ...

Aritmética de ponteiros e índices

Em C, o nome de um vetor é sinônimo da posição do primeiro elemento.

Assim, se declararmos

```
int v[10];
```

então `v` é o mesmo que `&v[0]`.

Desta forma, as atribuições

"`p = &v[0];`" e "`p = v;`"

são equivalentes.

Aritmética de ponteiros e índices

Como `v` é sinônimo do endereço do início do vetor, então

"`v[i]`" e "`*(v+i)`"

são duas maneiras equivalentes de nos referirmos ao mesmo elemento do vetor.

Assim, `*(v+1)` é `v[1]`, `*(v+2)` é `v[2]`, ...

Aritmética de ponteiros e índices

Reciprocamente, se `p` é um ponteiro e fizermos

"`p = &v[0];`" ou "`p = v;`"

então

`p[1]` é o mesmo que `v[1]`,
`p[2]` é o mesmo que `v[2]`, ...

Diferença entre ponteiros e nome de vetor

Enquanto um ponteiro é uma variável cujo conteúdo pode ser alterado escrevendo, por exemplo

"`p++;`" ou "`p = &v[3];`",

o nome de um vetor **não** é uma variável. Portanto, construções como

"`v++;`" ou "`v = v+2;`"

são ilegais.

Vetores como parâmetros

Como parâmetros formais de uma função,

```
char s[ ];
```

e

```
char *s;
```

são equivalentes.

Vetores como parâmetros

Como parâmetros formais de uma função,

```
char s[ ];
```

e

```
char *s;
```

são equivalentes.

Kernighan e Ritchie preferem a segunda pois
diz mais explicitamente que a variável é um apontador.

Vetores como parâmetros

Como parâmetros formais de uma função,

```
char s[ ];
```

e

```
char *s;
```

são equivalentes.

Kernighan e Ritchie preferem a segunda pois
diz mais explicitamente que a variável é um apontador.

Outro exemplo:

```
int main(int argc, char **argv);
```

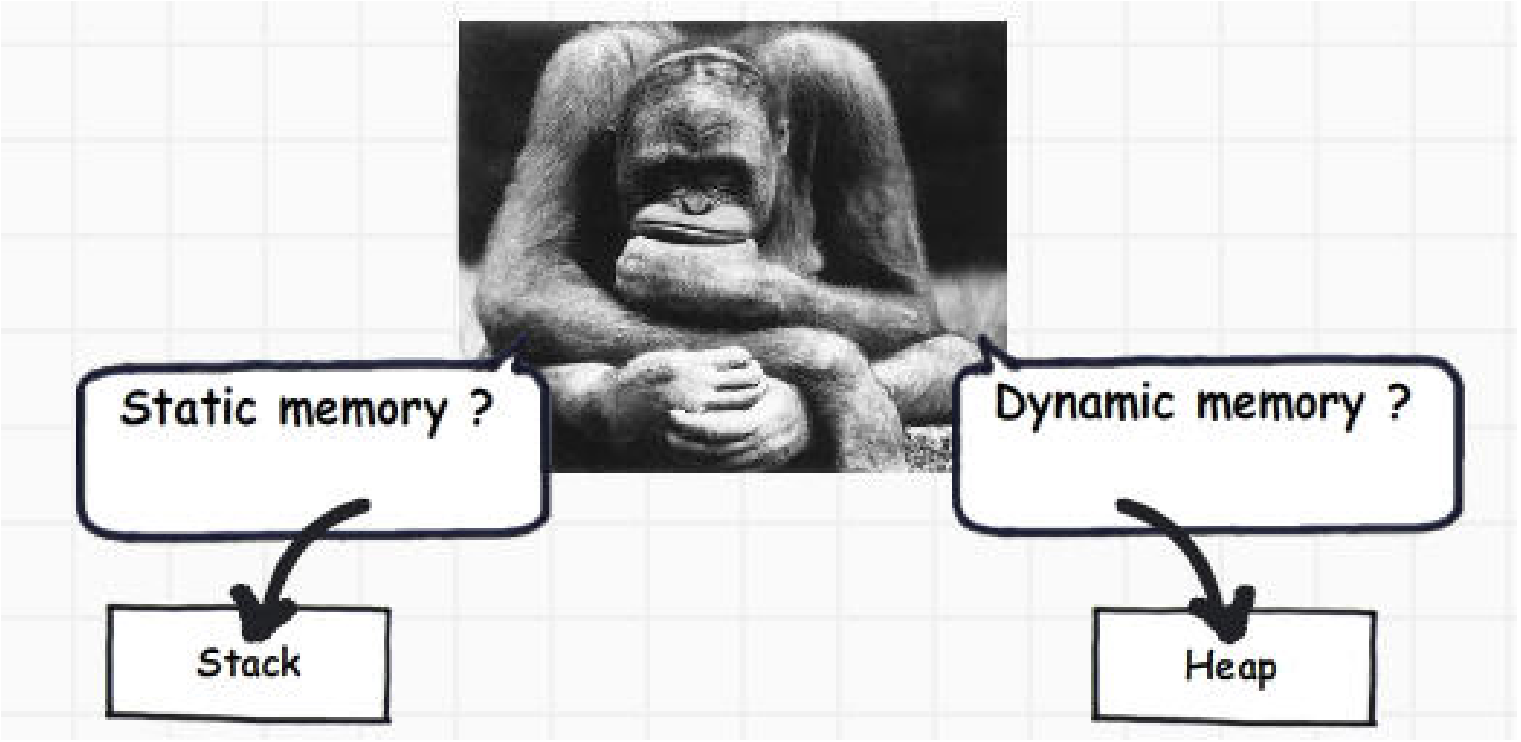
Alocação dinâmica de memória

PF Apêndice F

http://www.ime.usp.br/~pf/algoritmos/aulas/aloca.html

The C programming Language
Brian W. Kernighan e Dennis M. Ritchie
Prentice-Hall

Alocação dinâmica



Fonte: http://www.codeproject.com/

Alocação dinâmica

Às vezes, a quantidade de memória que
o programa necessita só se torna conhecida
durante a execução do programa.

Alocação dinâmica

Às vezes, a quantidade de memória que
o programa necessita só se torna conhecida
durante a execução do programa.

Para lidar com essa situação é preciso recorrer
à **alocação dinâmica de memória**.

Alocação dinâmica

Às vezes, a quantidade de memória que o programa necessita só se torna conhecida durante a execução do programa.

Para lidar com essa situação é preciso recorrer à **alocação dinâmica de memória**.

A alocação dinâmica é gerenciada pelas funções `malloc` e `free`, que estão na biblioteca `stdlib`:

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

malloc

A função `malloc` aloca um bloco de bytes consecutivos na memória e devolve o endereço desse bloco.

```
char *ptr;

ptr = malloc(1);
scanf("%c",ptr);
```

malloc

```
typedef struct {
    int dia,mes,ano;
} Data;

Data *d;

d = malloc(sizeof(Data));
```

malloc

Se `p` é ponteiro para uma estrutura, então

```
p->campo-da-estrutura
```

é uma abreviatura de

```
(*p).campo-da-estrutura
```

```
d->dia = 31; d->mes = 12; d->ano = 2008;
```

A memória é finita

Se `malloc` não consegue alocar mais espaço, então retorna `NULL`.

```
ptr = malloc(sizeof(Data));
if (ptr == NULL) {
   printf("Socorro!  malloc devolveu NULL!\n");
   exit(EXIT_FAILURE);
}
```

A memória é finita

É conveniente usarmos a função

```
void *mallocSafe (int nbytes) {
   void *ptr;

   ptr = malloc(nbytes);
   if (ptr == NULL) {
      printf("Socorro! malloc devolveu NULL!\n");
      exit(EXIT_FAILURE);
   }
   return ptr;
}
```

free

Fonte: http://www.zazzle.com.br/

free

A função `free` libera a memória alocada por `malloc`.

```
free(d);
```

Há pessoas que, por questões de segurança,
gostam de atribuir `NULL` a um ponteiro
depois da liberação de memória.

```
free(d);
d = NULL;
```

Vetores dinâmicos

```
int *v; int i, n;

printf("Digite o tamanho do vetor: ");
scanf("%d", &n);

v = mallocSafe(n*sizeof(int));

for (i = 0; i < n; i++)
   *(v+i) = i;    /* v[i] = i; */

for (i = 0; i < n; i++)
   printf("end. v[%d] = %p cont v[%d] = %d\n",
          i, (void*)(v+i), i, v[i]);
free(v);
```

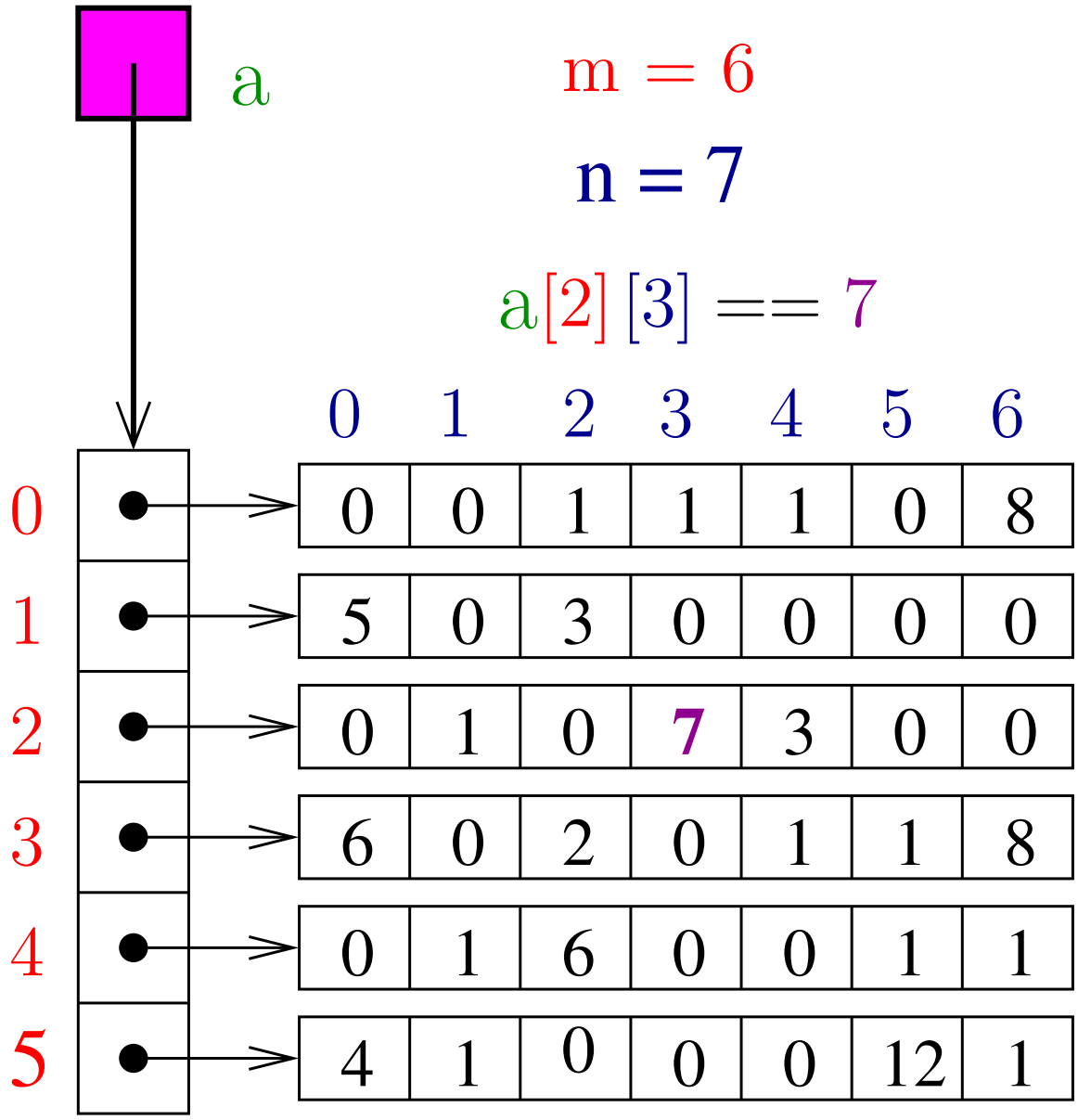
Matrizes dinâmicas

Matrizes bidimensionais são implementadas como vetores de vetores.

```
int **a;
int i;
a = mallocSafe(m * sizeof(int*));
for (i = 0; i < m; ++i)
   a[i] = mallocSafe(n * sizeof(int));
```

O elemento de `a` que está na linha `i` e coluna `j` é `a[i][j]`.

Matrizes dinâmicas



Liberação de memória de matrizes

Para liberarmos a memória alocada dinamicamente para uma matriz, devemos seguir os passos inversos aos da alocação, trocando `mallocSafe` por `free`.

Liberação de memória de matrizes

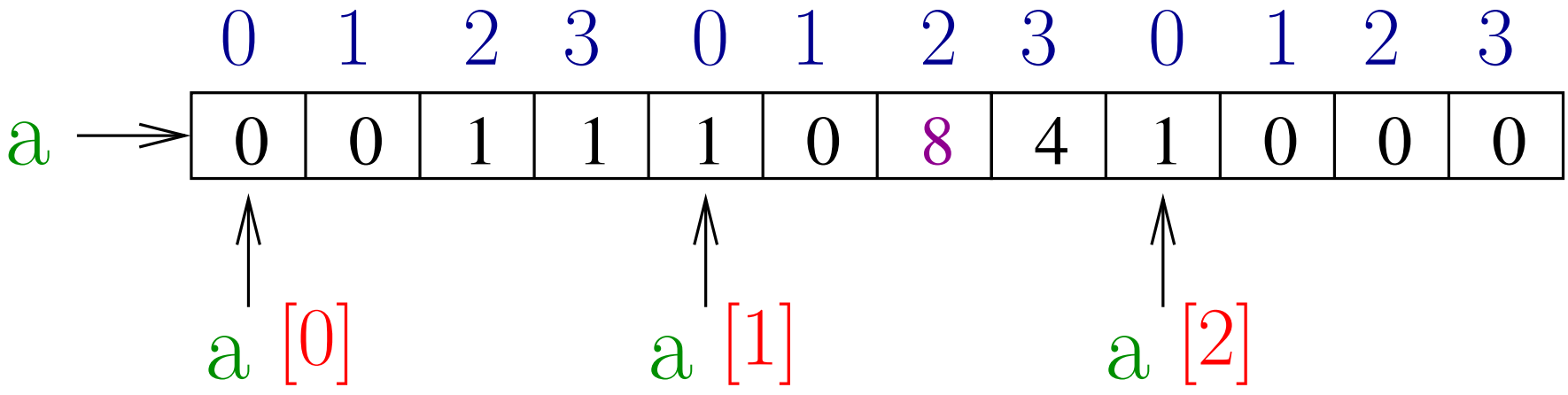
Para liberarmos a memória alocada dinamicamente para uma matriz, devemos seguir os passos inversos aos da alocação, trocando `mallocSafe` por `free`.

```
void freeMatrizInt(int **a, int m) {
   int i;

   for (i = 0; i < m; i++){
      free(a[i]);    /* libera a linha i */
      a[i] = NULL;
   }
   free(a);    /* libera vetor de ponteiros */
   a = NULL;
}
```

Matrizes automáticas

```
int a[3][4];
```

a [1] [2] == 8



Passagem de parâmetros

Suponha que temos os protótipos de funções

```
void f(int **m);
int g(int m[][64]);
```

e as declarações

```
int **a;
int m[16][64];
```

Passagem de parâmetros

Suponha que temos os protótipos de funções

```
void f(int **m);
int g(int m[][64]);
```

e as declarações

```
int **a;
int m[16][64];
```

então temos que

```
f(a);          /* ok */
i = g(a);      /* erro */
i = g(m);      /* ok */
f(m);          /* erro */
```