Melhores momentos

AULA 6

Conceitos

Endereços: a memória é um vetor e o índice desse vetor onde está uma variável é o endereço da variável.

Com o operador `&` obtemos o endereço de uma variável.

Exemplos:

- `&i` é o endereço de `i`
- `&ponto` é o endereço da estrutura `ponto`
- `&v[2]` é o endereço de `v[2]`

Conceitos

Ponteiros: são variáveis que armazenam endereços.

Exemplos:

```
int *p;  /* ponteiro para int*/
char *q;  /* ponteiro para char*/
double *r;  /* ponteiro para double*/
```

Conceitos

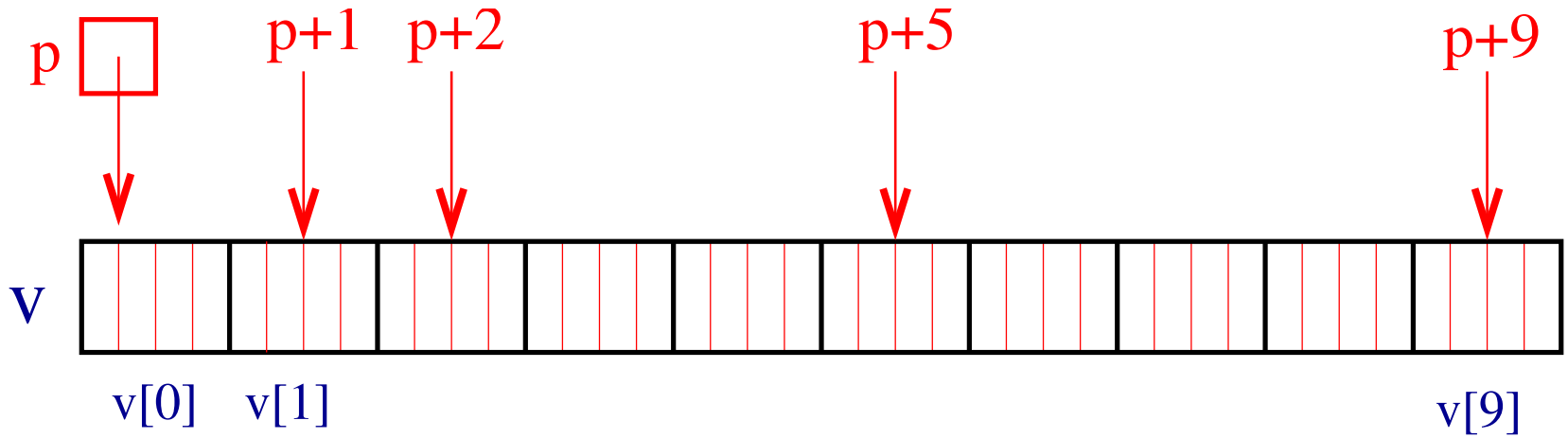
Dereferenciação: Se `p` aponta para a variável `i`, então `*p` é sinônimo de `i`.

Exemplo:

```
p = &i;   /* p aponta para i/
(*p)++;   é o mesmo que   i++;
```

Conceitos

Aritmética de ponteiros: se `p` é um apontador para um `int` e o seu contéudo é 64542, então `p+1` é 64546, pois um `int` ocupa 4 bytes (no meu computador...).



Conceitos

Vetores e ponteiros: o nome de um vetor é sinônimo do endereço da posição inicial do vetor.

Exemplo:

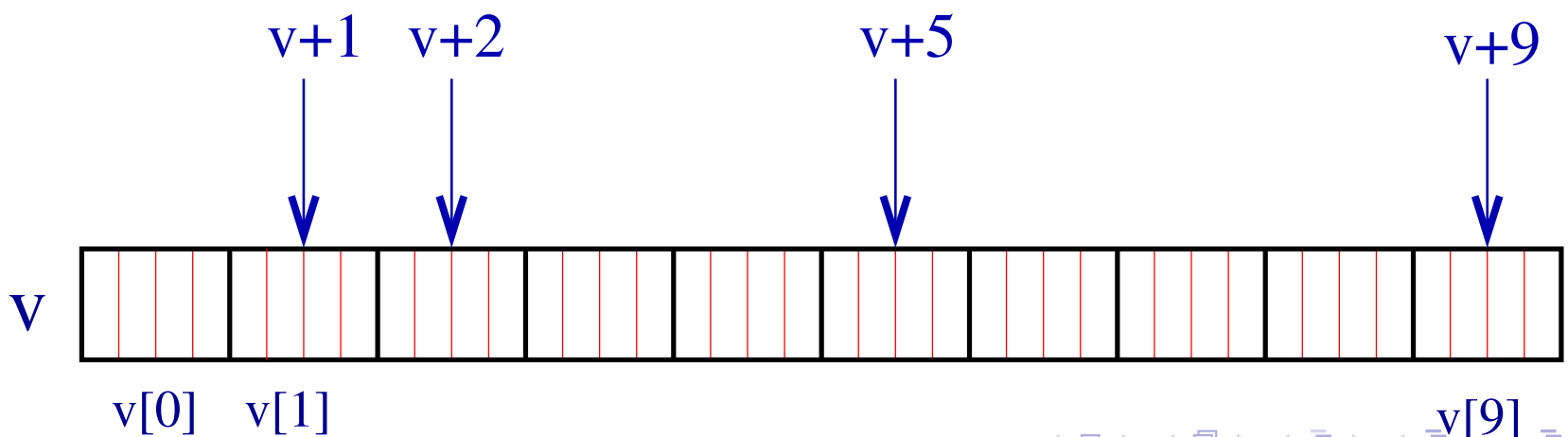
```
int v[10];
```

v é sinônimo de &v[0]

v+1 é sinônimo de &v[1]

v+2 é sinônimo de &v[2]

. . .



AULA 7

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

As 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**.

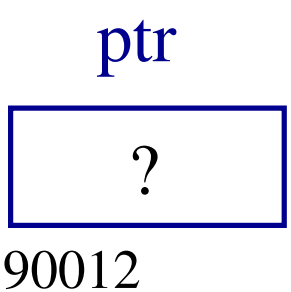
A alocação dinâmica é gerenciada pelas funções `malloc` e `free`, que estão na biblioteca `stdlib`

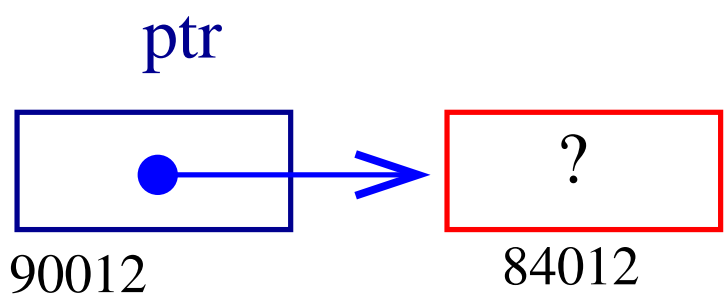
```
#include <stdlib.h>
```

malloc

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

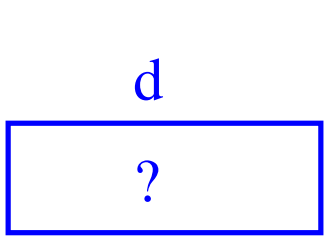
```
char *ptr;
ptr = malloc(1);
scanf("%c",ptr);
```

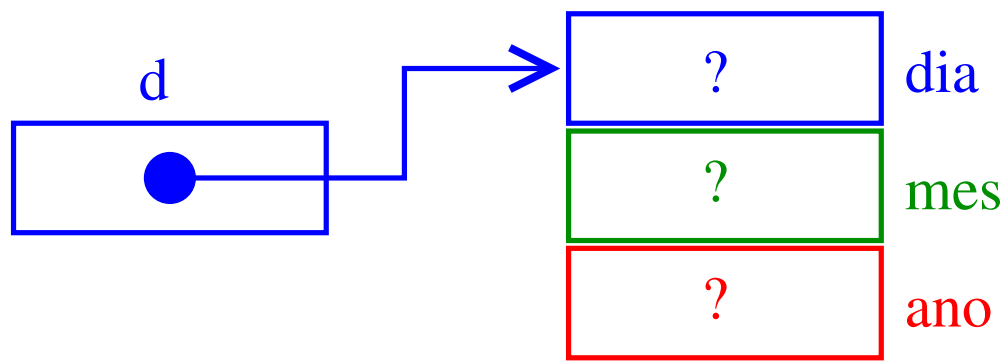




malloc

```
typedef struct {
    int dia,mes,ano;
} Data;
Data *d;
d = malloc (sizeof(Data));
```





malloc

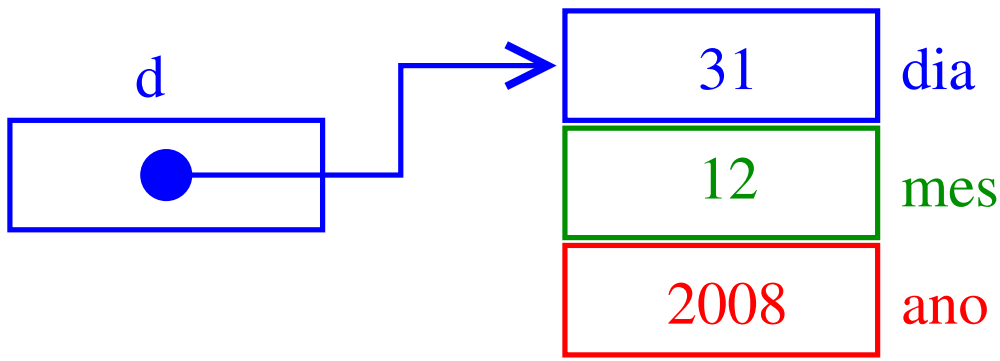
Se `p` é ponteiro para uma estrutura então

```
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 e devolve 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

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 liberaçõa de memória

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

Vetores dinamicamente

```
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;

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);
```

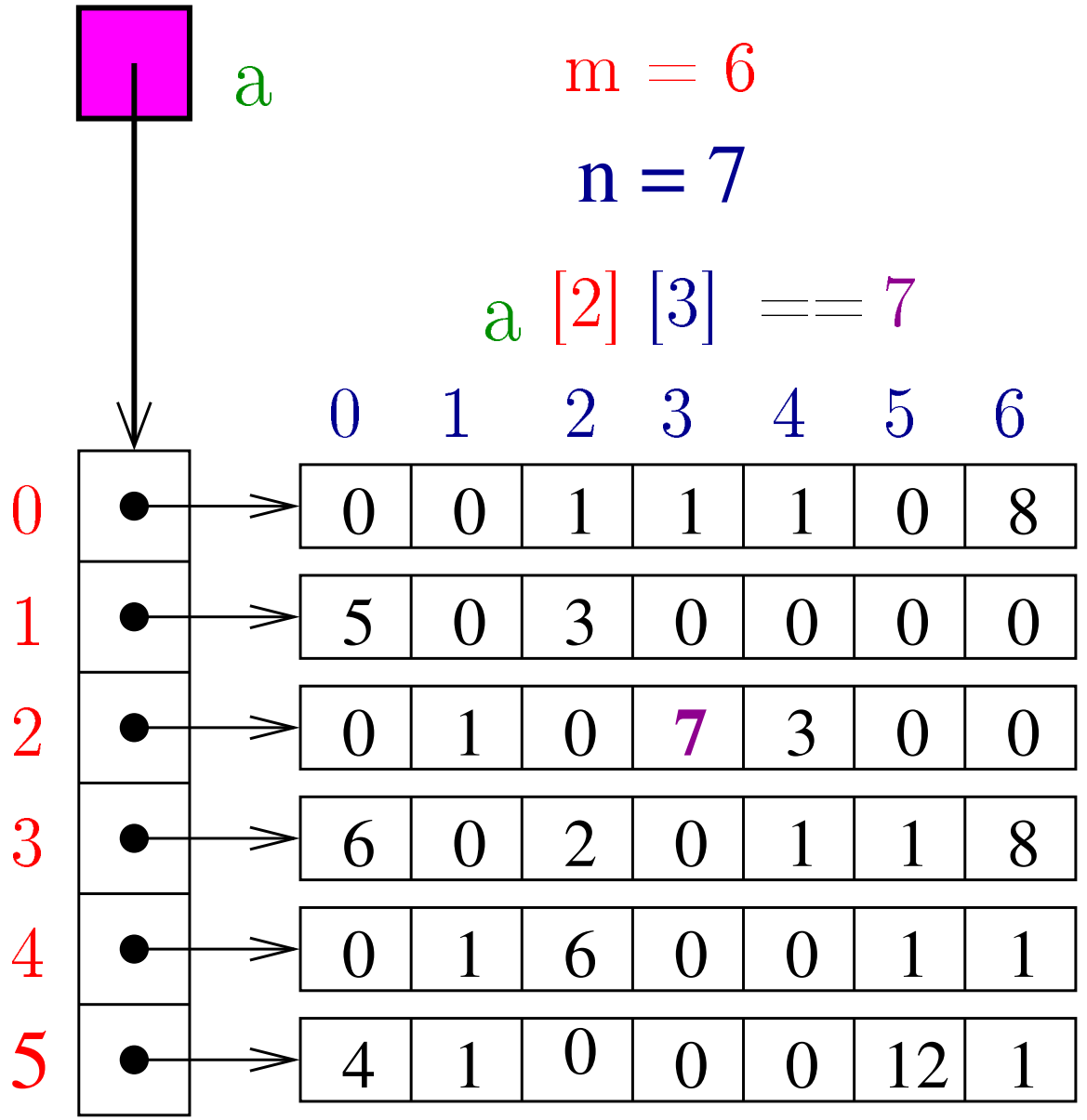
Matrizes dinâmicas

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

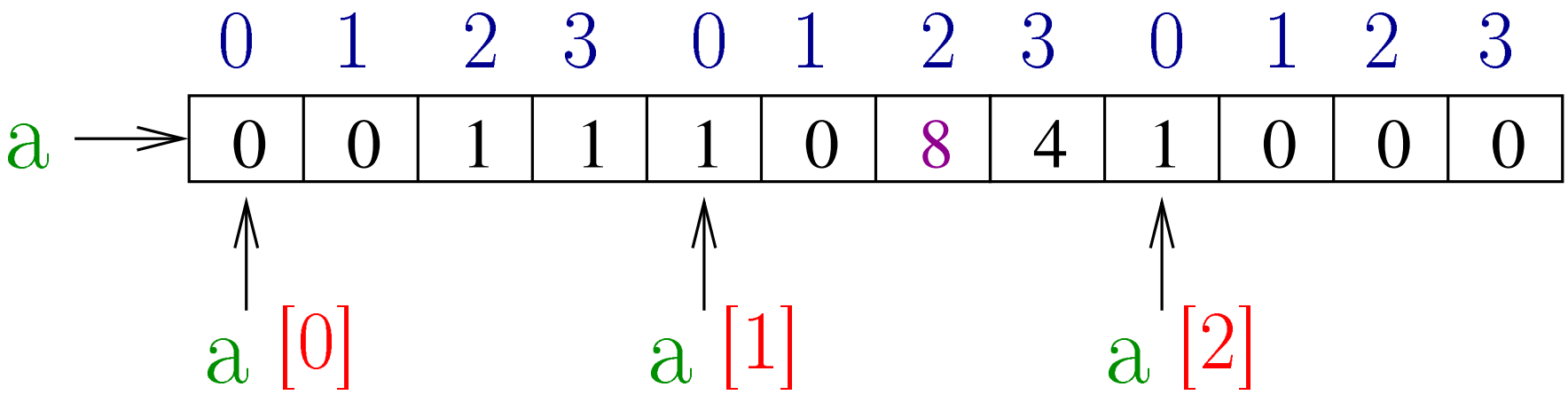
Para liberarmos a memória alocada dinamicamente para uma matriz devemos seguir os passos inversos aos da alocação trocando `mallocSafe` por `free`.

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

   for (i = 0; i < m; i++){
       free(a[i]); /* liberar 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];
```

então temos que

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