

AULA 6

- ▶ registros e estruturas
- ▶ endereços e ponteiros

Registros e Structs

PF Apêndice E

<http://www.ime.usp.br/~pf/algoritmos/aulas/stru.html>

Registros e structs

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

Na linguagem C, registros são conhecidos como **structs**.

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



Nomes de estruturas

É uma boa idéia dar um **nome**, digamos **data**, à estrutura.

Nosso exemplo ficaria melhor assim

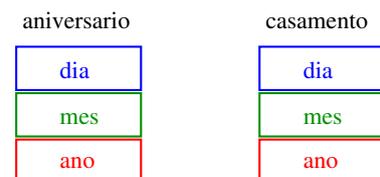
```
struct data {
    int dia;
    int mes;
    int ano;
};
struct data aniversario;
```



Estruturas e tipos

Um declaração de **struct** define um tipo.

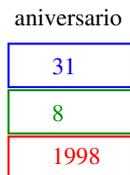
```
struct data aniversario;
struct data casamento;
```



Campos de uma estrutura

É fácil atribuímos valores aos campos de uma estrutura:

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



Estruturas e typedef

Para não repetir “`struct data`” o tempo todo podemos definir uma abreviatura via `typedef`:

```
struct data{
    int dia;
    int mes;
    int ano;
};
typedef struct data Data;
Data aniversario;
Data casamento;
```

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

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[4];
```

Endereços

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

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
```

< > < > < > < > < > < > < >

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

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



< > < > < > < > < > < > < >

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

< > < > < > < > < > < > < >

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` tem valor diferente de `NULL` então

`*p`

é o objeto apontado por `p`.



< > < > < > < > < > < > < >

Vetores e endereços

Suponha que p é um ponteiro para um inteiro

```
int *p;
```

então a atribuição

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

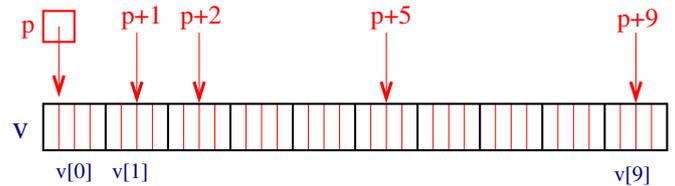
faz com p contenha o endereço de $v[0]$



Navigation icons: back, forward, search, etc.

Aritmética de ponteiros

Se p aponta para um elemento do vetor, estão $p+1$ 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 .

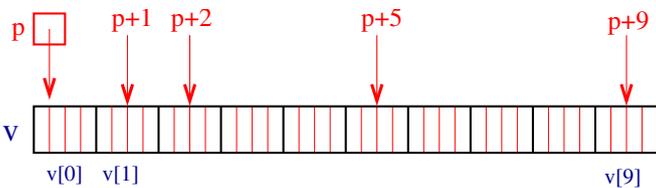


Assim, $*(p+1)$ é $v[1]$, $*(p+2)$ é $v[2]$,...

Navigation icons: back, forward, search, etc.

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 $v[2]$,...

Navigation icons: back, forward, search, etc.

Aritmética de ponteiros e índices

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

Assim, se declararmos

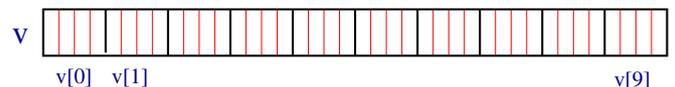
```
int v[10];
```

então v é o mesmo que $\&v[0]$.

Desta forma, as atribuições

```
“p = &v[0];” e “p = v;”
```

são equivalentes.



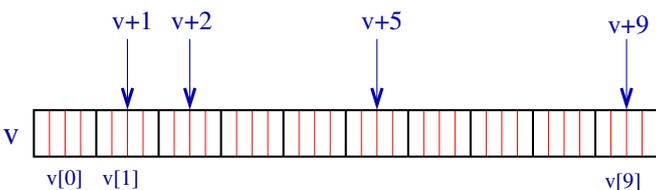
Navigation icons: back, forward, search, etc.

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 $v[2]$,...

Navigation icons: back, forward, search, etc.

Aritmética de ponteiros e índices

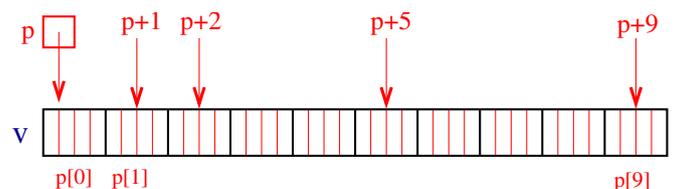
Reciprocamente, se p é um ponteiro e fizermos

```
“p = &v[0];” ou “p = v;”
```

então

$p[1]$ é o mesmo que $v[1]$,

$p[2]$ é o mesmo que $v[2]$,...



Navigation icons: back, forward, search, etc.

