Structs, Ponteiros e Arquivos

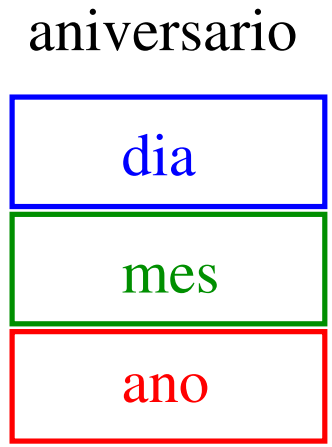
Prof. Maurício Dias

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;
```

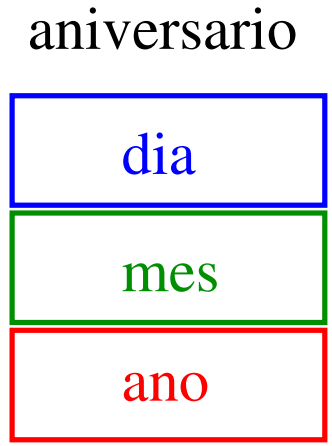


Nomes de estruturas

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

Nosso exemplo ficaria melhor assim

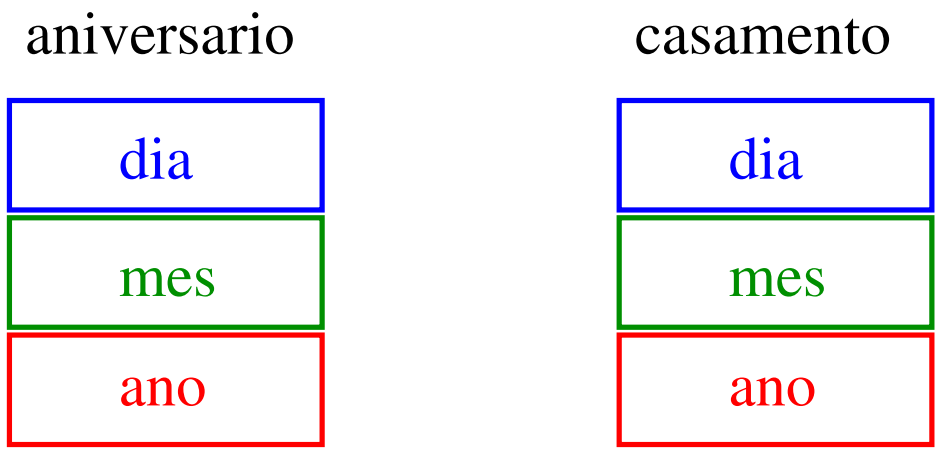
```
struct data {
    int dia;
    int mes;
    int ano;
} ;

struct data aniversario;
```



Estruturas e tipos

Um declaração de `struct` define um tipo.

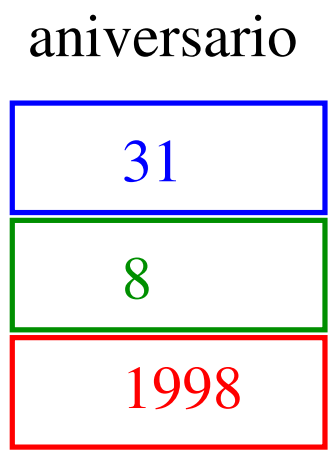
```
struct data aniversario;
struct data casamento;
```



Campos de uma estrutura

É fácil atribuirmos 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

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
```

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`

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

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, ponteiros 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 e' 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;
}
```

Chamada da função

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

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);
```

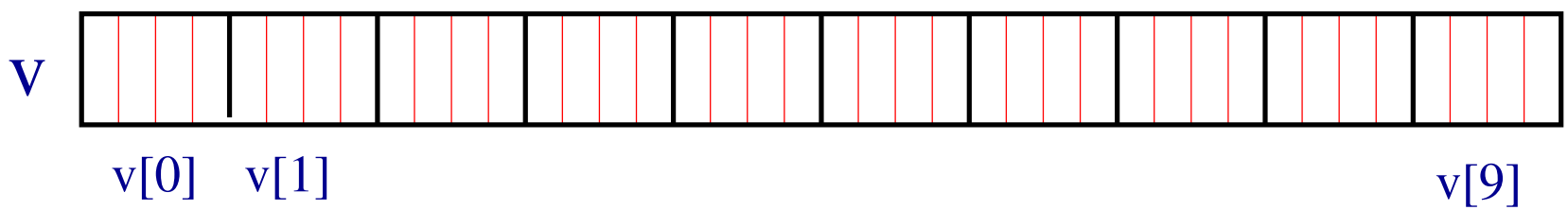
Vetores e endereços

Em C, existe uma relação muuuiito 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

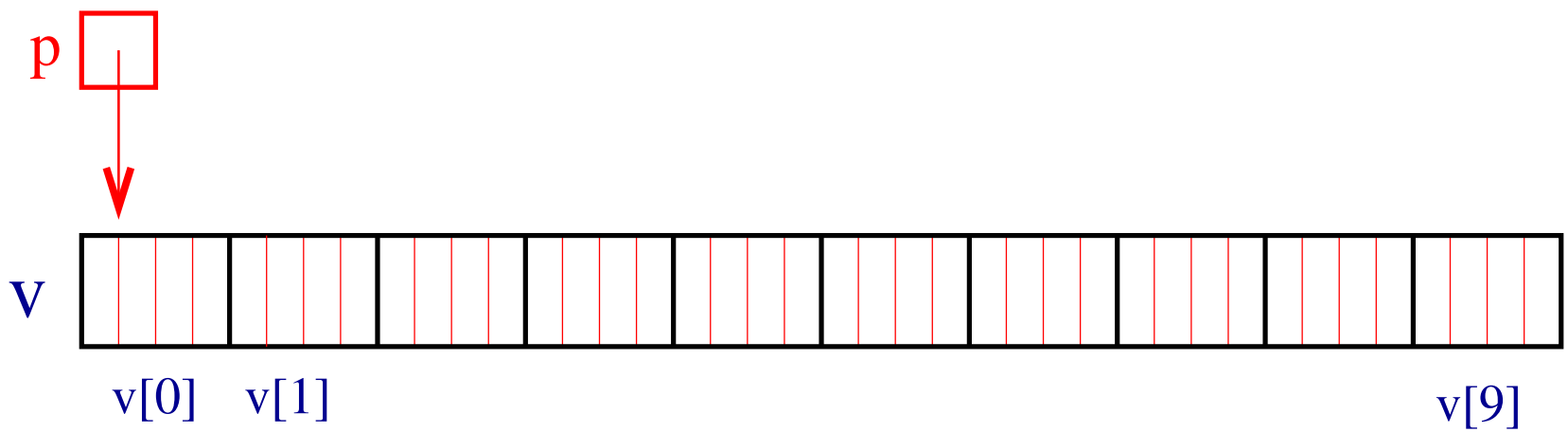
Suponha que `p` é um ponteiro para um inteiro

```
int *p;
```

então a atribuição

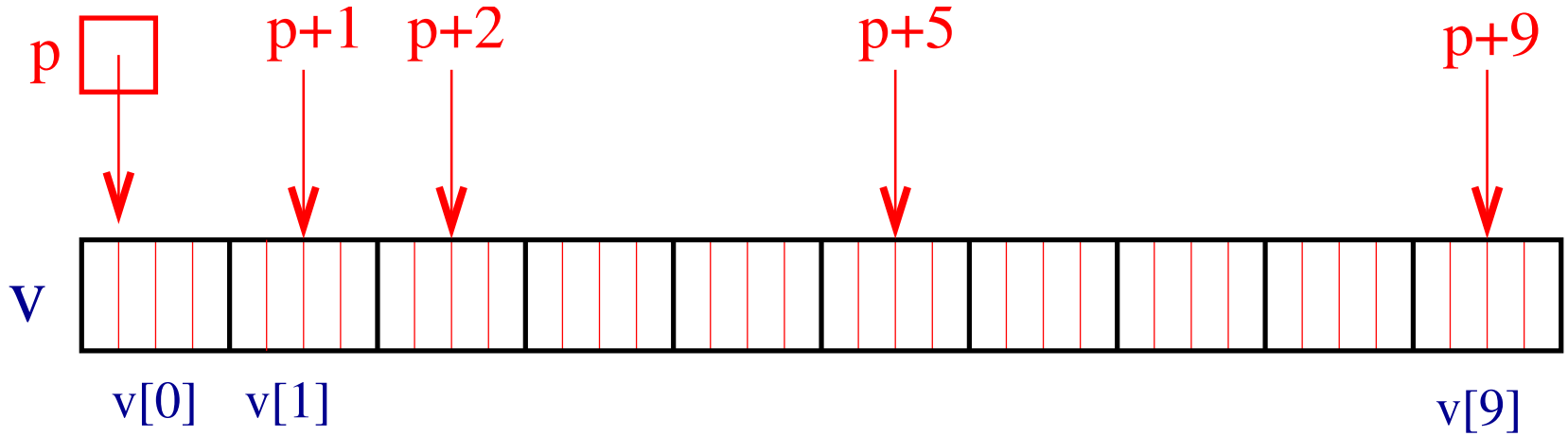
```
p = &v[0];
```

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



Aritmética de ponteiros

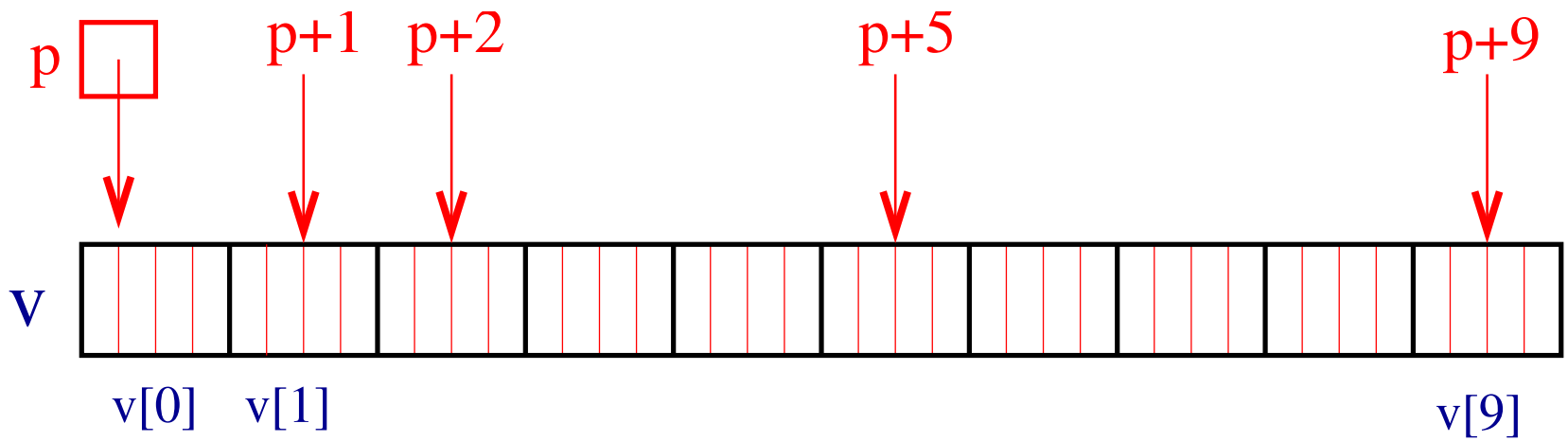
Se `p` aponta para um elemento do vetor, estão `p+1` aponta para o elemento seguinte, `p+i` aponta para o `i`-ésimo elemento depois de `p`, `p-i` para o `i`-ésimo elemento antes de `p`.



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

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

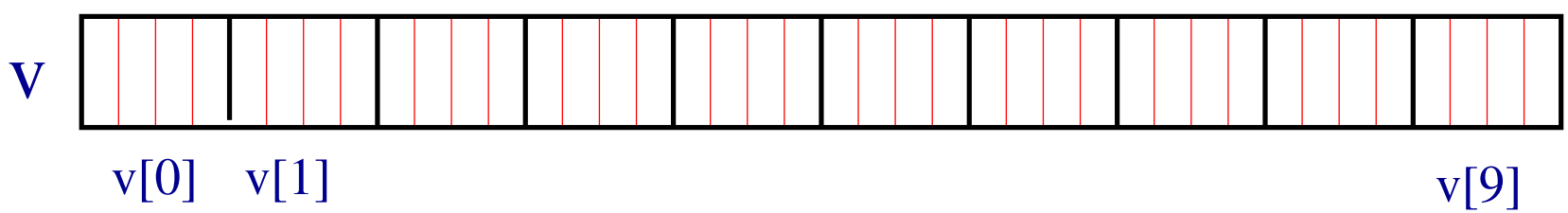
```
int v[10];
```

então `v` é o mesmo que `&v[0]`.

Desta forma, as atribuições

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

são equivalentes.

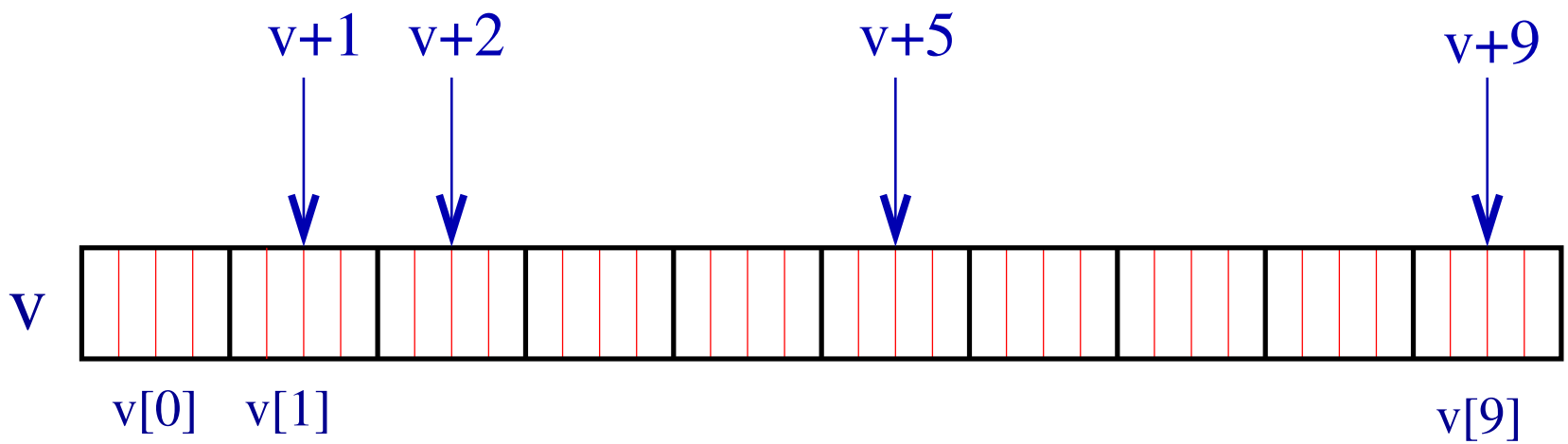


Aritmética de ponteiros e índices

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]`,...

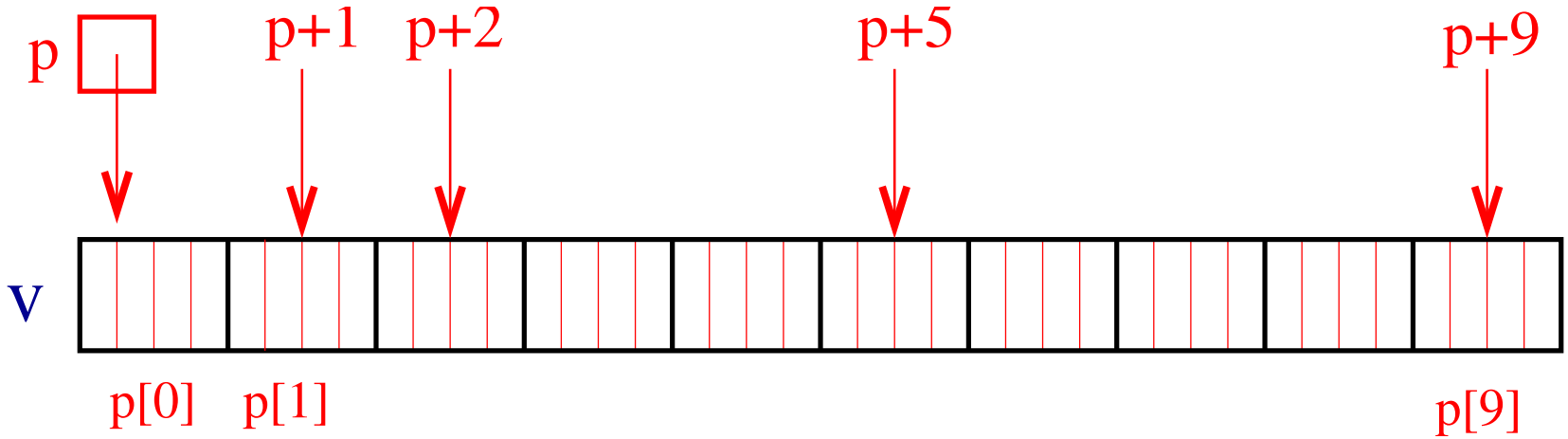
Aritmética de ponteiros e índices

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

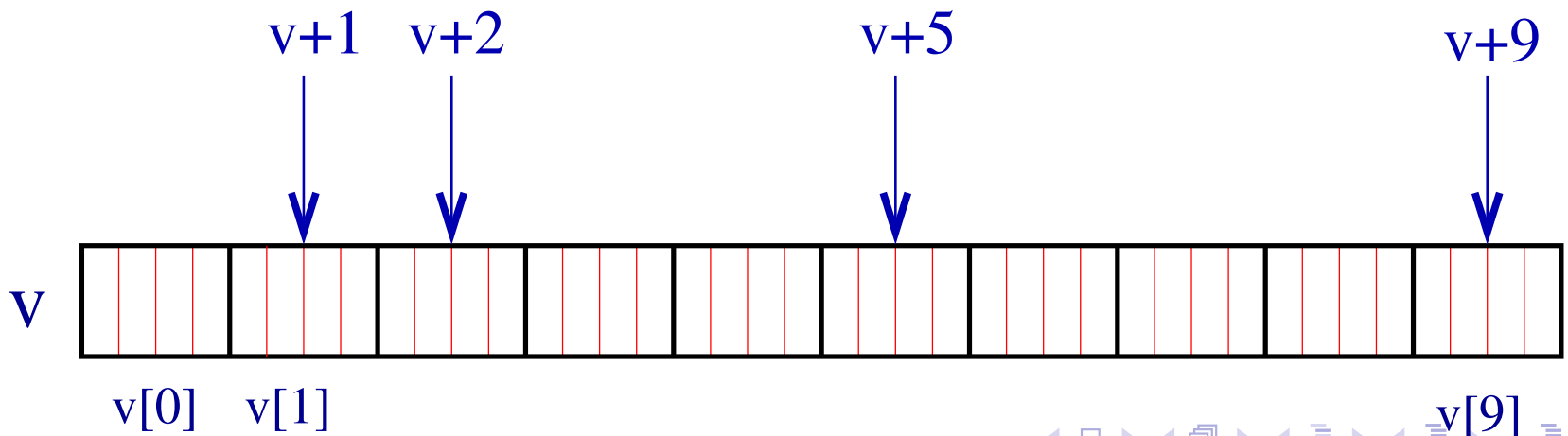
Enquanto um ponteiro é uma variável que podemos alterar o seu conteúdo escrevendo, por exemplo

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

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

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

são ilegais.



Vetores como parâmetros

Como parâmetros formais de uma função,

```
char s[ ];
```

e

```
char *s;
```

são equivalentes. O Kernighan e Ritchie preferem a segunda pois diz mais explicitamente que a variável é um apontador.
Outro exemplo

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

Conceitos na aula de hoje

- 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 na aula de hoje

- Ponteiros: são variáveis que armazenam endereços.

  Exemplos:

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

- 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 na aula de hoje

- Aritmética de ponteiros: se `p` é um apontador para um `double` e o seu conteúdo é 64542, então `p+1` é 64550, pois um `double` ocupa 8 bytes (no meu computador...).
- Vetores e ponteiros: o nome de um vetor é sinônimo do endereço da posição inicial do vetor.

  Exemplo:

  ```
  char nome[124];
  ```

  `nome` é sinônimo de `&nome[0]`

  `nome+1` é sinônimo de `&nome[1]`

  `nome+2` é sinônimo de `&nome[2]`

  . . .

Manipulação de Arquivo em C

- Existem dois tipos possíveis de acesso a arquivos na linguagem C : sequencial (lendo um registro após o outro) e aleatório (posicionando-se diretamente num determinado registro)
- Os arquivos em C são denominados STREAM
- Um STREAM é associado a um arquivo por uma operação de abertura do arquivo e, a partir da associação, todas as demais operações de escrita e leitura podem ser realizadas

Manipulação de Arquivo em C

- A tabela abaixo apresenta as principais funções da linguagem C para manipulação de arquivos

Função	Ação
`fopen()`	Abre um arquivo
`Fclose ()`	Fecha um arquivo
`putc() e fputc()`	Escreve um caractere em um arquivo
`getc() e fgetc()`	Lê um caractere de um arquivo
`fseek()`	Posiciona em um registro de um arquivo
`fprintf()`	Efetua impressão formatada em um arquivo
`fscanf()`	Efetua leitura formatada em um arquivo
`feof()`	Verifica o final de um arquivo
`fwrite()`	Escreve tipos maiores que 1 byte em um arquivo
`fread()`	Lê tipos maiores que 1 byte de um arquivo

Manipulação de Arquivo em C

- O sistema de entrada e saída do ANSI C, sendo composto por uma série de funções, cujos protótipos estão reunidos em stdio.h
- Todas as funções relacionadas anteriormente trabalham com o conceito de ponteiro de arquivo, sendo definido usando o comando typedef
- Esta definição também está no arquivo stdio.h, e um ponteiro de arquivo pode ser declarado da seguinte maneira:

```
FILE   *Arquivo;
```

Manipulação de Arquivo em C

- Pela declaração do ponteiro anterior, passa a existir uma variável de nome Arquivo, que é ponteiro para um arquivo a ser manipulado
- O ponteiro de arquivo une o sistema de E/S a um buffer e não aponta diretamente para o arquivo em disco, contendo informações sobre o arquivo, incluindo nome, status (aberto, fechado e outros) e posição atual sobre o arquivo

Abrindo um Arquivo

- A função que abre um arquivo em C é a função ***fopen()***, que devolve o valor NULL (nulo) ou um ponteiro associado ao arquivo, devendo ser passado para função o nome físico do arquivo e o modo como este arquivo deve ser aberto

```
Arquivo = fopen ("texto.txt","w");
```

Abrindo um Arquivo

- De acordo com a instrução anterior, está sendo aberto um arquivo de nome “texto.txt”, habilitado apenas para escrita (w-write)
- Por exemplo, pode-se codificar a instrução de abertura de arquivo da seguinte maneira:

```
if ((Arquivo = fopen("texto.txt","w")) == NULL)  {
  printf("\n Arquivo TEXTO.TXT não pode ser aberto : TECLE ALGO");
  getch();
}
```

Abrindo um Arquivo

- Além do modo de escrita, a linguagem C permite o uso de alguns valores padronizados para o modo de manipulação de arquivos, conforme mostra a tabela abaixo:

Modo	**Significado**
"r"	Abre um arquivo texto para leitura. O arquivo deve existir antes de ser aberto.
"w"	Abrir um arquivo texto para gravação. Se o arquivo não existir, ele será criado. Se já existir, o conteúdo anterior será destruído.
"a"	Abrir um arquivo texto para gravação. Os dados serão adicionados no fim do arquivo ("append"), se ele já existir, ou um novo arquivo será criado, no caso de arquivo não existente anteriormente.
"rb"	Abre um arquivo binário para leitura. Igual ao modo "r" anterior, só que o arquivo é binário.
"wb"	Cria um arquivo binário para escrita, como no modo "w" anterior, só que o arquivo é binário.
"ab"	Acrescenta dados binários no fim do arquivo, como no modo "a" anterior, só que o arquivo é binário.
"r+"	Abre um arquivo texto para leitura e gravação. O arquivo deve existir e pode ser modificado.
"w+"	Cria um arquivo texto para leitura e gravação. Se o arquivo existir, o conteúdo anterior será destruído. Se não existir, será criado.
"a+"	Abre um arquivo texto para gravação e leitura. Os dados serão adicionados no fim do arquivo se ele já existir, ou um novo arquivo será criado, no caso de arquivo não existente anteriormente.
"r+b"	Abre um arquivo binário para leitura e escrita. O mesmo que "r+" acima, só que o arquivo é binário.
"w+b"	Cria um arquivo binário para leitura e escrita. O mesmo que "w+" acima, só que o arquivo é binário.
"a+b"	Acrescenta dados ou cria uma arquivo binário para leitura e escrita. O mesmo que "a+" acima, só que o arquivo é binário

Fechando um Arquivo

- Para o esvaziamento da memória de um arquivo é utilizada a função ***fclose()***, que associa-se diretamente ao nome lógico do arquivo (STREAM):

```
fclose (Arquivo);
```

Gravando e lendo Dados em Arquivos

- Existem várias funções em C para a operação de gravação e leitura de dados em arquivos. Abaixo seguem algumas:
  - ***putc()*** ou ***fputc()***: Grava um único caracter no arquivo
  - ***fprintf()*** : Grava dados formatados no arquivo, de acordo com o tipo de dados (float, int, ...). Similar ao printf, porém ao invés de imprimir na tela, grava em arquivo
  - ***fwrite()*** : Grava um conjunto de dados heterogêneos (struct) no arquivo
  - ***fscanf()***: retorna a quantidade variáveis lidas com sucesso

Sintaxe das funções para gravação

- Grava o conteúdo da variável caracter no arquivo

```
putc (caracter, arquivo);
```

- Grava dados formatados no arquivo, de acordo com o tipo de dados (float, int, ...)

```
fprintf(arquivo,  "formatos", var1, var2 ...);
```

- Grava um conjunto de dados heterogêneos (struct) no arquivo

```
fwrite (buffer, tamanhoembytes, quantidade, ponteirodearquivo);
```

- Retorna a quantidade variáveis lidas com sucesso

```
fscanf(arquivo, "formatos", &var1, &var2 ...);
```

Um programa usando o fscanf

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <cstdlib>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {
     FILE *arq_cliente;
     char var_sexo, var_arquivo_aux, nomecli[50];
     int cd_cli, vl_idade,  indice = 0;
     float vl_limite_credito;
     arq_cliente = fopen("CLIENTE.TXT", "r");

     if (arq_cliente == NULL) {
         printf("\nArquivo CLIENTE.TXT nao pode ser aberto.");
         printf("\nOcorreu um Erro Grave ! Use alguma tecla para finalizar !");
         getch();
     }else {
        var_arquivo_aux = fscanf(arq_cliente, "%d %c %s %d %f",&cd_cli, &var_sexo
                                                ,&nomecli, &vl_idade, &vl_limite_credito);
        while (var_arquivo_aux != EOF)  {
               indice = indice + 1;
               printf("\n Dados do %d § cliente : \n ", indice);
               printf("\n Codigo do Cliente...: %d  Sexo..: %c", cd_cli, var_sexo);
               printf("\n Nome do Cliente.................: %s ", nomecli);
               printf("\n Idade..........: %d  Credito....: %8.2f", vl_idade, vl_limite_credito);
               printf("\n---------------------------------------------------------------  [Tecle algo] !");
               getch();
               var_arquivo_aux = fscanf(arq_cliente, "%d %c %s %d %f",&cd_cli, &var_sexo
                                                ,&nomecli, &vl_idade, &vl_limite_credito);
       }
       fclose (arq_cliente);
       printf("\n *** FIM  : [Tecle algo] !");
       getch();
     }
}
```

Lendo e Gravando Estruturas

- Além da manipulação de arquivos do tipo texto, pode-se ler e escrever estruturas maiores que 1 byte, usando as funções ***fread()*** e ***fwrite()***, conforme as sintaxes a seguir:

```
fread (buffer, tamanhoembytes, quantidade, ponteirodearquivo)
fwrite(buffer, tamanhoembytes, quantidade, ponteirodearquivo)
```

Lendo e Gravando Estruturas

- O ***buffer*** é um endereço de memória da estrutura de onde deve ser lido ou onde devem ser escritos os valores (fread() e fwrite(), respectivamente)
- O ***tamanhoembytes*** é um valor numérico que define o número de bytes da estrutura que deve ser lida/escrita
- A ***quantidade*** é o número de estruturas que devem ser lidas ou escritas em cada processo de fread ou fwrite
- O ***ponteirodearquivo*** é o ponteiro do arquivo de onde deve ser lida ou escrita uma estrutura

Lendo e Gravando Estruturas

- Normalmente é necessário manipular arquivos por meio de estruturas de dados ou arquivos de estruturas (struct)
- Podemos por exemplo falar num arquivo de **CLIENTES**, onde cada cliente possui **NOME**, **RG**, **ENDERECO** E **TELEFONE**

Lendo e Gravando Estruturas

- A função ***sizeof*** retorna a quantidade de bytes de um determinado tipo ou variável
- Tal função é importante para que o programa de manipulação de arquivos possa saber se ainda existem registros para serem lidos
- Por exemplo, enquanto o retorno da instrução abaixo for igual a 1, o programa continua lendo registros:

```
retorno = fread(&Vcli, sizeof(struct Tcliente), 1, cliente);
```

Posicionando em um registro

- Por meio da linguagem C não é possível saber qual é a posição de cada registro no arquivo
- Em outras linguagens, a movimentação em registros é feita por meio de funções que fazem a leitura da linha do registro
- Em C esta posição pode ser calculada pelo tamanho do registro

Posicionando em um registro

- Não é possível, como em outras linguagens, pedir para que se posicione no segundo, terceiro ou último registro
- Para isso, programador em C deve saber o tamanho em bytes de cada registro, e posicionar-se de acordo com este tamanho.
- A função ***seek()***, apresentada logo abaixo movimenta-se de byte em byte

```
seek(<referência_ao_arquivo>, <n>, <modo>);
```

Posicionando em um registro

- O parâmetro `<n>` indica quantos bytes devem ser avançados ou retrocedidos
- O exemplo a seguir posiciona-se no quarto registro do arquivo de cliente
- Observe que é utilizada uma função auxiliar – a função ***sizeof()*** que indica quantos bytes possui o registro a ser inserido (ou a estrutura definida para o registro)

```
fseek(Arquivo_de_Cliente, 4 * sizeof(Cliente), SEEK_SET);
```

Posicionando em um registro

- Neste caso o tipo Cliente, que é o registro, foi utilizado para indicar o tamanho de cada registro
- Multiplicando-se o valor retornado por quatro obtém-se o local do quarto registro do arquivo. Caso o local (o registro) solicitado não exista não será feito o posicionamento e o registro atual continuará sendo o mesmo

Posicionando em um registro

- Outros parâmetros usados pela função seek()

  - **SEEK_SET** - Parte do início do arquivo e avança <n> bytes
  - **SEEK_END** - Parte do final do arquivo e retrocede <n> bytes
  - **SEEK_CUR** - Parte do local atual e avança <n> bytes