Fundamentos de Programação
Vetores e funções

Dainf - UTFPR

Profa. Leyza B. Dorini
Prof. Bogdan T. Nassu

Vetores em funções

Vetores em C são **sempre** passados por referência.

Atenção!

Portanto, ao passar um vetor como parâmetro, se ele for alterado dentro da função, as alterações ocorrerão no próprio vetor e não em uma cópia.

Mas por que é sempre passado por referência?

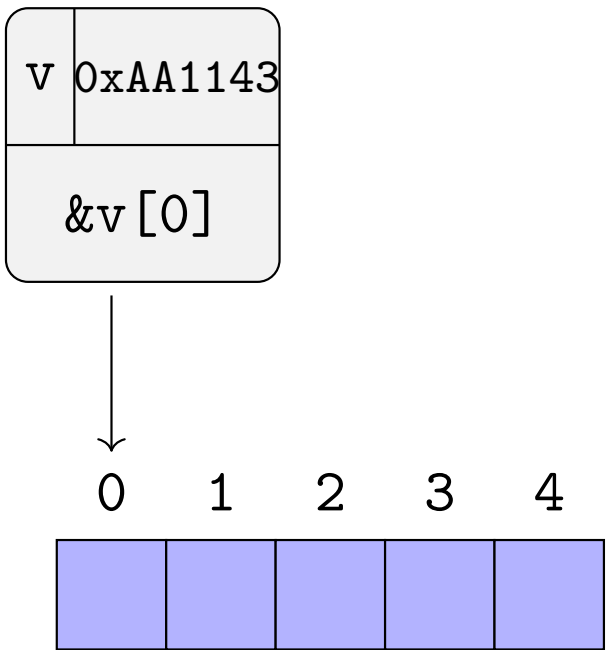
Vetores e ponteiros

Vetores e ponteiros!

Atenção!

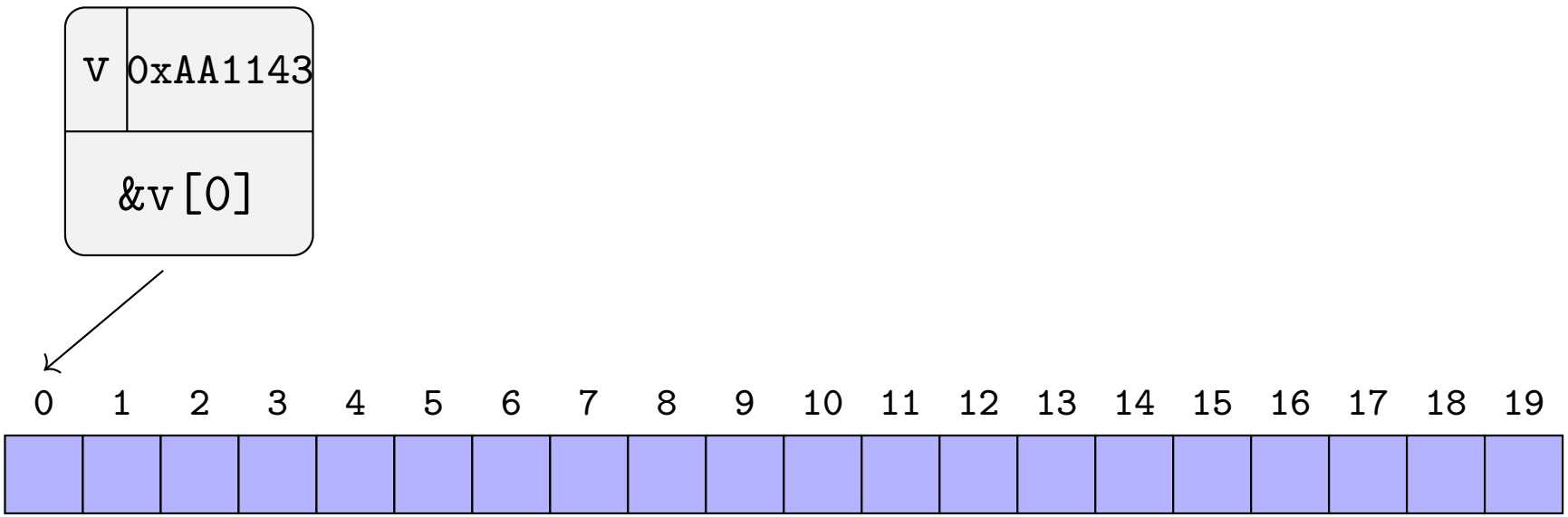
Uma "variável vetor" é um ponteiro que armazena o **endereço** da primeira posição de memória alocada.

Suponha a declaração `int v[5];` Neste caso, a variável (ponteiro) `v` contém o endereço de memória do início do vetor (ou seja, do elemento `v[0]`).



Vetores e ponteiros!

A quantidade de memória alocada depende do tipo do vetor! Por exemplo, para a declaração `int v[5]`, teríamos o seguinte cenário:



ou seja, teríamos a alocação de 20 bytes (dado que cara inteiro ocupa 4 bytes), sendo que o endereço do primeiro seria atribuído para o ponteiro.

Ponteiros e vetores

Esta "dupla identidade" entre ponteiros e vetores é a responsável pelo fato de vetores serem sempre passados por referência e pela inabilidade da linguagem em detectar acessos fora dos limites de um vetor.

Freshmen in my university getting mad because of a SegFault

Vetores são sempre passados por referência

Portanto, lembre-se:

Ao passar um vetor como parâmetro para uma função, seu conteúdo pode ser alterado dentro da função (pois estamos passando, na realidade, o endereço de início do espaço alocado para o vetor).

Passando vetores por parâmetro

Como passar um vetor por parâmetro?

Ao invocar a função, observe que não passamos o endereço de `v` (`v`), mas sim o próprio conteúdo de `v`:

```
1  int main ()
2  {
3      int v[10] = {1,2,3,4,5,6,7,8,9,10};
4      imprimeVetor(v, 10);
5      mudaVetor(v, 10);
6      imprimeVetor(v, 10);
7
8      return 0;
9  }
```

Isso se deve ao fato que `v` é, na verdade, uma variável ponteiro contendo o endereço da primeira posição de memória do vetor!

Qual a sintaxe para o vetor no protótipo da função?

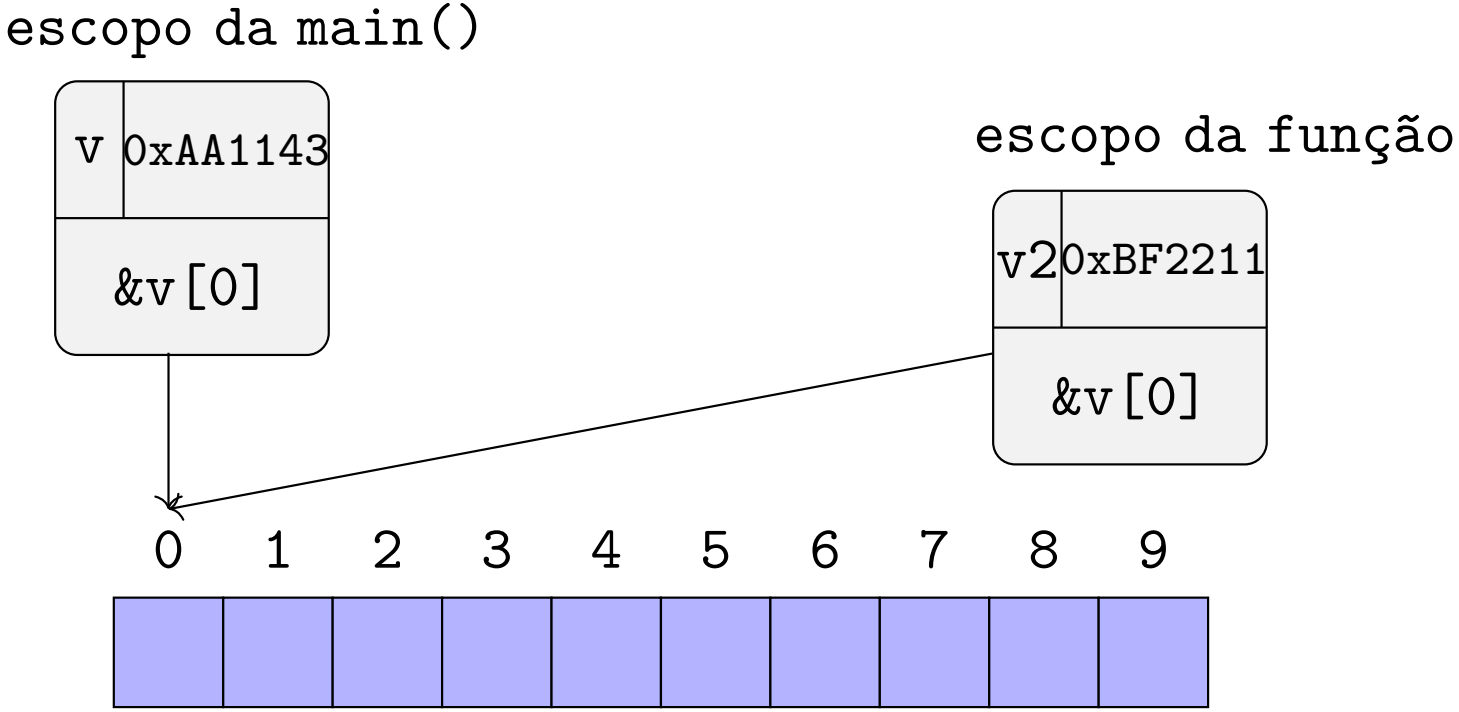
Formas de criar uma função que recebe um vetor como parâmetro:

```
void imprimeVetor(int *v, int tam)
{
    (...) //É a mais "popular".
}

void imprimeVetor(int v[], int tam)
{
    (...) //É equivalente à anterior.
}

void imprimeVetor(int v[N])
{
    (...) //Pouco usada, só aceita vetores do mesmo tamanho!
}
```

Estes são exemplos! Podemos ter outros tipos de retorno, outros identificadores, mais parâmetros...

Passando vetores por parâmetro

Ao passar um vetor por parâmetro, a função terá um ponteiro (local) apontando para o endereço de memória que foi originalmente alocado para o vetor! Isso é uma convenção: o compilador não sabe o que o parâmetro passado representa!



Exemplo

```
1  int main ()
2  {
3      int v[10] = {1,2,3,4,5,6,7,8,9,10};
4      imprimeVetor(v, 10); //função que imprime um vetor
5      mudaVetor(v, 10); /* função que multiplica o conteúdo
6                           de cada posição por 2*/
7      imprimeVetor(v,10);
8
9      return 0;
10 }
```

Como vetores são passados por referência, o conteúdo inicial foi modificado dentro da função `mudaVetor()`:
Saída:

```
1   2   3   4   5   6   7   8   9  10
2   4   6   8  10  12  14  16  18  20

Process returned 0 (0x0)   execution time : 0.017 s
```

Acessando valores recebidos por parâmetro

Como acessar um vetor recebido por parâmetro?

Dentro da função, independentemente do protótipo utilizado, podemos acessar o vetor com a mesma notação que temos usado até agora.

```
1 void mudaVetor(int v[])
2 {
3     int i;
4     for(i=0; i<TAM; i++)
5         v[i]=v[i]*2;
6 }
7
```

Entretanto, também é possível acessar usando a notação de ponteiros!

Importante!

Vetores e ponteiros

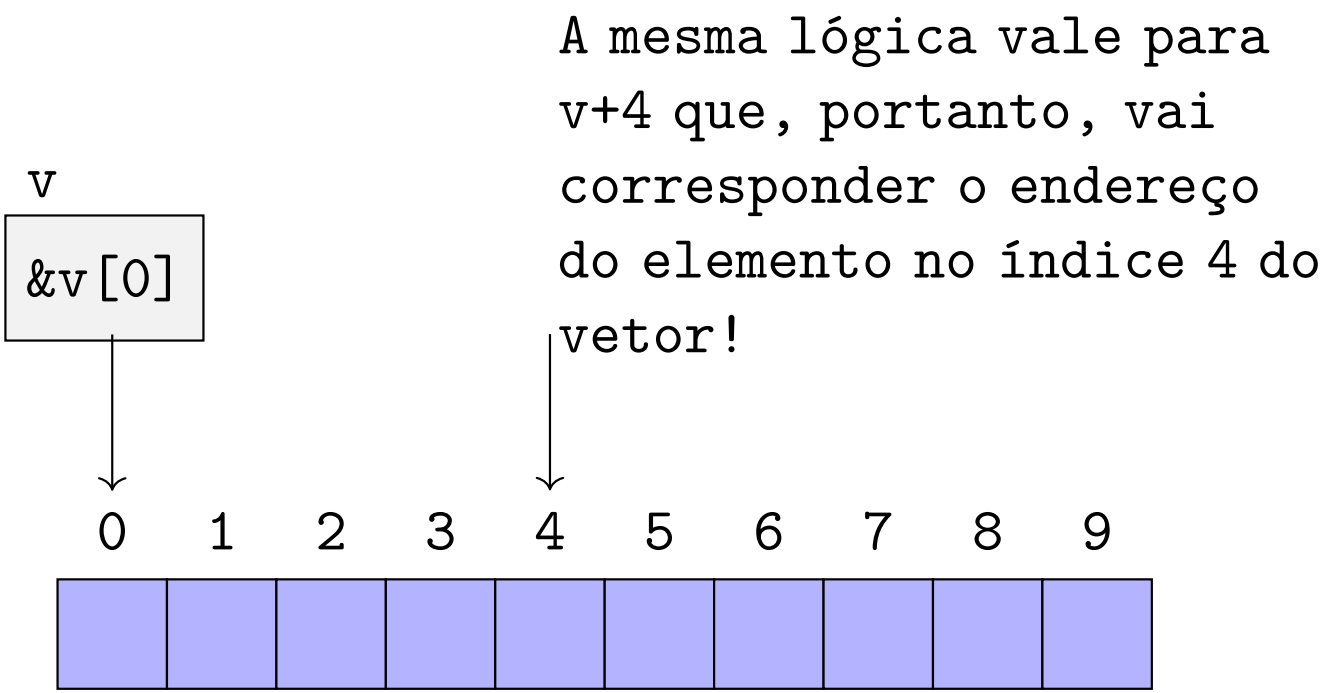
Escrever `v[i]` é equivalente a escrever `*(v+i)`

A operação de indexação corresponde a deslocar o ponteiro ao longo dos elementos alocados ao vetor, o que pode ser feito de duas formas:

- Usando o operador de indexação: `v[4]`.
- Usando aritmética de endereços: `*(v+4)`.

Importante!

Como `v` armazena o endereço de `v[0]`, a operação `v+1` vai resultar no próximo endereço de memória (por isso a importância de as posições alocadas serem contíguas). Mas cuidado: o próximo endereço depende do tipo do ponteiro (se for `int`, isso significa somar 4 bytes no endereço):



Ao passar vetores por parâmetro, a função precisa saber o seu tamanho!

Vetores com tamanho definido por uma constante

Nos exemplos anteriores, observe que **passamos como parâmetro, além do vetor, o seu tamanho!** Se o tamanho do vetor fosse definido por uma constante, por exemplo, esse argumento não seria necessário! Exemplo para a função `mudaVetor()`.

```
1  #define TAM 10
2
3  void mudaVetor(int v[TAM])
4  {
5      int i;
6      for(i=0; i<TAM; i++)
7          v[i]=v[i]*2;
8  }
```

Entretanto, esta forma só funciona com vetores de tamanho específico.

Vetores com tamanho definido por uma constante

Contudo, é uma boa prática considerar o tamanho como parâmetro de entrada. Dessa forma, a função fica mais genérica (trabalha com qualquer tamanho e a operação sendo executada pela função pode ser realizada em apenas parte do vetor).

```
1  #define TAM 10
2
3  void mudaVetor(int v[], int n){
4      int i;
5      for(i=0; i<n; i++)
6          v[i]=v[i]*2;
7  }
8
9  int main(){
10     int v[TAM] = {5,6,5,7,4,1,2,3,6,7};
11     mudaVetor(v, 5); //apenas parte do vetor é alterado.
12     mudaVetor(v, TAM); //todo o vetor é acessado
13
14     return 0;
15 }
```

Como retornar um vetor?

Como retornar um vetor?

`int [] fooFunction ();` → ERRADO!

`int [N] fooFunction ();` → ERRADO!

`int* fooFunction ();` → Compila, mas a não ser que você esteja usando alocação dinâmica, está ERRADO!

Você pode declarar um vetor dentro da função, mas não pode retorná-lo (ele deixa de existir ao final da função)!

Como retornar um vetor por parâmetro?

Até aprendermos alocação dinâmica, passe o vetor "resposta" por parâmetro.

Suponha, por exemplo, uma função que armazena em um novo vetor (`output`) o dobro dos elementos do vetor de entrada (`input`).

```
void testa(int input[], int output[], int tam)
{
    int i;
    for(i=0; i<tam; i++)
        output[i] = input[i]*2;
}
```

Note que, ao invés de criarmos o vetor `output` dentro da função e retorná-lo, ele é recebido como parâmetro de entrada (ou seja, é declarado na própria função que invocou `testa()`).

Exemplos

Exemplo: retorno do maior valor em um vetor

```
1  int maiorValor(int vet[], int tam) {
2      int i, max;
3
4      max = vet[0];
5
6      for (i = 1; i < tam; i++) {
7          if (vet[i] > max)
8              max = vet[i];
9      }
10     return max;
11 }
12
13 int main() {
14     int v[9] = {10, 80, 5, -10, 45, -20, 1, 2, 10},
15         max;
16
17     max = maiorValor(v, 9);
18
19     printf("Maior valor: %d\n", max);
20     return 0;
21 }
```

Exemplo: retorno do maior e do menor valor em um vetor

```
1  void minmax(int vet[], int tam, int *min, int *max) {
2      int i;
3      *min = vet[0];
4      *max = vet[0];
5      for (i = 1; i < tam; i++) {
6          if (vet[i] < *min)
7              *min = vet[i];
8          if (vet[i] > *max)
9              *max = vet[i];
10     }
11 }
12
13 int main() {
14     int v[] = {10, 80, 5, -10, 45, -20, 100, 200, 10};
15     int min, max;
16     minmax(v, 9, &min, &max);
17     printf("Menor valor: %d\n", min);
18     printf("Maior valor: %d\n", max);
19     return 0;
20 }
```

Exemplo - Copiando os elementos pares de um vetor em outro

```
1  int copPares(int v1[],int v2[],int t_v1) {
2      int i, t_v2=0;
3      for (i = 0; i < t_v1; i++){
4          if (v1[i]%2 == 0){
5              v2[t_v2] = v1[i];
6              t_v2++; /*necessário para saber quantos elementos
       válidos o v2 possui*/
7          }
8      }
9      return t_v2;
10 }
11 int main() {
12     int v1[5] = {10, 80, 5, 20, 45}, v2[5], tam_v2, i;
13     tam_v2 = copPares(v1, v2, 5);
14     printf("Existem %d pares:\n", tam);
15     for(i=0; i<tam_v2; i++)
16         printf("%d ", v2[i]);
17     return 0;
18 }
```

Exemplo - Copiando os elementos pares de um vetor em outro (retorno do tamanho por referência)

```
1  void copPares(int v1[],int v2[],int t_v1, int *t_v2) {
2      int i;
3      *t_v2=0;
4      for (i = 0; i < t_v1; i++){
5          if (v1[i]%2 == 0){
6              v2[*t_v2] = v1[i];
7              (*t_v2)++; /*necessário para saber quantos elementos
       válidos o v2 possui*/
8          }
9      }
10 }
11 int main() {
12     int v1[5] = {10, 80, 5, 20, 45}, v2[5], tam, i;
13
14     copPares(v1, v2, 5, &tam);
15     printf("Existem %d pares:\n", tam);
16     for(i=0; i<tam; i++)
17         printf("%d ", v2[i]);
18     return 0;
```

Como diferenciar ponteiros e vetores?

Qual a diferença?

Funções que recebem parâmetros por referência:

```
int func1 (int* parametro, int outro_parametro);
void func2 (char* parametro);
```

Funções que recebem vetores como parâmetros:

```
int func1 (int* parametro, int tamanho);
void func2 (char* parametro);
```

Mas não é a mesma coisa?!

Qual a diferença?

Funções que recebem parâmetros por referência:

```
int func1 (int* parametro, int outro_parametro);
void func2 (char* parametro);
```

Funções que recebem vetores como parâmetros:

```
int func1 (int* parametro, int tamanho);
void func2 (char* parametro);
```

Mas não é a mesma coisa?!

As duas versões de `func1` recebem um ponteiro para `int` e um inteiro. As duas versões de `func2` recebem um ponteiro para `char`. Você só pode diferenciar através de comentários, contexto, forma de uso da função...

Exemplo (vetor)

```
1  void fooFunction (int* foo, int bar)
2  {
3      int i;
4      for (i = 0; i < bar; i++)
5          foo [i] = 0;
6  }
7
8  int main ()
9  {
10     int um_vetor [10];
11     fooFunction (um_vetor, 10);
12     return (0);
13 }
```

Exemplo (ponteiro)

Repare como o protótipo da função não mudou, mas os parâmetros e a forma de usá-los é outra.

```
1  void fooFunction (int* foo, int bar)
2  {
3      *foo = bar;
4  }
5
6  int main()
7  {
8      int um_int;
9      fooFunction (&um_int, 10);
10     return (0);
11 }
```

Exemplo (bug!)

Acessar um inteiro passado por referência como um vetor pode resultar em um *buffer overflow* ou um acesso inválido à memória. `&um_int` é como um vetor de 1 posição!

```
1  void fooFunction (int* foo, int bar)
2  {
3      int i;
4      for (i = 0; i < bar; i++)
5          foo [i] = 0;
6  }
7
8  int main()
9  {
10     int um_int;
11     fooFunction (&um_int, 10);
12     return (0);
13 }
```

Exemplo (não é um bug!)

Acessar um vetor como se fosse um inteiro passado por referência funciona! No exemplo abaixo, a função coloca o valor 10 na primeira posição do vetor.

```
1  void fooFunction (int* foo, int bar)
2  {
3      *foo = bar;
4  }
5
6  int main()
7  {
8      int um_vetor [10];
9      fooFunction (um_vetor, 10);
10     return (0);
11 }
```