

Ponteiros e Alocação Dinâmica de Memória

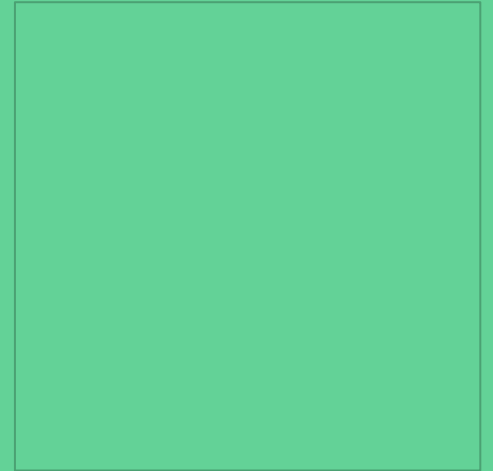
Prof. Joaquim Quinteiro Uchôa
Profa. Juliana Galvani Gregghi
Profa. Marluce Rodrigues Pereira
Profa. Paula Christina Cardoso
Prof. Renato Ramos da Silva



Roteiro

- Ponteiros
- Uso de ponteiros
- Alocação dinâmica de memória
- Redimensionamento de vetores
- Cuidados com o uso de ponteiros

Ponteiros



Relembrando Variáveis

Variáveis ocupam espaço da memória e possuem um endereço para acesso!

```
char letra = 'A';  
int num = 25;  
string texto = "oi";
```

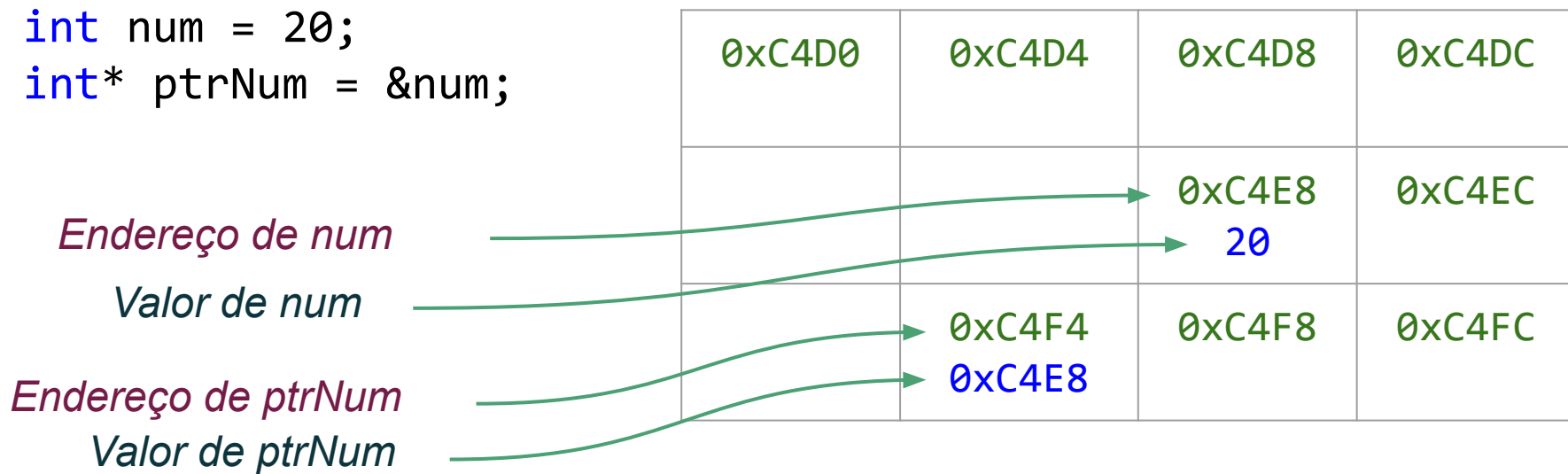
0001	0002	0003	0004
0005	0006	0007	0008

letra	👉	[0001]	⇒	&letra = 1
num	👉	[0002-0005]	⇒	&num = 2
texto	👉	[0006-0008]	⇒	&texto = 6

Ponteiros

Ponteiros são variáveis que armazenam endereços de memória (ou seja, o endereço de outras variáveis).

```
int num = 20;  
int* ptrNum = &num;
```



Uso de Ponteiros

Existem diversos usos para ponteiros em programação, destacando-se:

- Acesso indireto a valores, muitas vezes sem necessidade de uma variável
- Casting dinâmico de variáveis - (acesso de uma variável de um tipo como se fosse de outro)
- Alocação e gerenciamento dinâmicos de memória



Indireção

É possível utilizar ponteiros para acessar os valores de outras variáveis:

```
int p = 5;  
int* ptr = &p;  
cout << *ptr << endl; // vai imprimir 5  
*ptr = 8;  
cout << p << endl; // vai imprimir 8
```



Indireção

É possível utilizar ponteiros para acessar os valores de outras variáveis:

```
int p = 5;  
int* ptr = &p;  
cout << *ptr << endl; // vai imprimir 5  
*ptr = 8;  
cout << p << endl; // vai imprimir 8
```

referência

derreferência

Cuidados com uso - i

Assim, como com variáveis, ponteiros devem ser inicializados ou receberem valores antes de serem usados:

```
int aux;  
int* auxPtr;  
cout << "TRECHO PERIGOSO\n";  
cout << " Valor de aux:" << aux << endl;  
cout << "Endereço de aux:" << &aux << endl;  
cout << "Valor de auxPtr:" << auxPtr << endl;  
cout << "Valor apontado por auxPtr:" << *auxPtr << endl;
```



Cuidados com uso - ii

Várias falhas surgem ao atribuir valores para ponteiros não inicializados ou inicializados de forma incorreta:

```
int* auxPtr;  
cout << "TRECHO MUITO PERIGOSO\n";  
cout << "Valor de auxPtr:" << auxPtr << endl;  
cout << "Valor apontado por auxPtr:"  
    << *auxPtr << endl;  
auxPtr = 240; // erro grave!!! que variável está na  
              // região de memória sendo acessada???
```



Cuidados com uso - iii

Exemplo seguro:

```
int aux;  
int* auxPtr;  
aux = 21;  
auxPtr = &aux;  
cout << "TRECHO SEGURO\n";  
cout << " Valor de aux:" << aux << endl;  
cout << "Endereço de aux:" << &aux << endl;  
cout << "Valor de auxPtr:" << auxPtr << endl;  
cout << "Valor apontado por auxPtr:" << *auxPtr << endl;
```



Uso do valor NULL

Quando se trabalha com ponteiros, é muito comum inicializá-los com NULL ou 0, para indicar que não apontam para nenhuma variável.

```
int* myPtr = NULL;  
(...)  
if (myPtr) { // só é verdadeiro se myPtr não for NULL  
    (...)  
}
```

NULL

Obs: em C++, tanto faz usar NULL ou 0, entretanto o primeiro é mais adequado, por ser mais claro e significativo. C++ também possui a expressão `nullpointer`, que tem a vantagem de ter a palavra reservada **nullptr**.

Exemplo de uso de NULL

```
int vetor[10];
for (int i = 0; i < 10; ++i)
    cin >> vetor[i];
int procurado, posicao;
int* ptPos = NULL;
cin >> procurado;
int i = 0;
while ( (i < 10) and (ptPos == NULL) ) {
    if (vetor[i] == procurado) {
        posicao = i;
        ptPos = &posicao;
    }
    i++;
}
if (ptPos != NULL)
    cout << "Encontrado na posicao: " << *ptPos << "\n";
```



Uso de ponteiros



Declarando ponteiros - i

No que diz respeito ao estilo, ponteiros podem ser declarados de três formas distintas:

```
int* ptr;  
int *ptr;  
int * ptr;
```

A terceira forma é totalmente não recomendada, uma vez que piora a legibilidade do código, aumentando a confusão do mesmo com o operador de multiplicação.

Declarando ponteiros - ii

```
int* ptr;  
int *ptr;
```

A segunda forma é preferida por aqueles que preferem destacar a variável. A primeira forma é preferida pelos que preferem destacar o tipo. Entretanto, a primeira forma não pode ser usada para declarar vários ponteiros em uma única linha:

```
int* ptr1, ptr2; // ptr1 é ponteiro, ptr2 não  
int *ptr1, *ptr2; // ptr1 e ptr2 são ponteiros
```



Cuidados com operador *

É recomendável o uso de () para evitar confusões entre o operador de indireção e a multiplicação:


```
int p = 10;
int q = 20;
int* ptr1 = &p;
int* ptr2 = &q;
cout << *ptr1 << endl;
cout << *ptr2 << endl;
cout << *ptr1 *ptr2 << endl; // trecho com erro!
cout << (*ptr1) * (*ptr2) << endl; // trecho claro e sem erros
```



Vetores e Ponteiros - i

Vetores são ponteiros constantes:

```
int vetor[10];  
for (int i=0; i<10; i++){  
    cin >> vetor[i];  
}  
int* ptr1 = (int*) &vetor;  
int* ptr2 = vetor;  
cout << vetor << endl;  
cout << &vetor << endl;  
cout << &vetor[1] << endl;  
cout << ptr1 << endl;  
cout << ptr2 << endl;
```



casting, enxergando o endereço do vetor como um ponteiro para inteiro

Vetores e Ponteiros - i

Vetores são ponteiros constantes:

```
int vetor[10];
for (int i=0; i<10; i++){
    cin >> vetor[i];
}
int* ptr1 = (int*) &vetor;
int* ptr2 = vetor;
cout << vetor << endl;
cout << &vetor << endl;
cout << &vetor[1] << endl;
cout << ptr1 << endl;
cout << ptr2 << endl;
```

```
[joukim@harpia tmp]$ ./ptvec
9 8 7 6 5 4 3 2 1 0
0x7fff05536010
0x7fff05536010
0x7fff05536014
0x7fff05536010
0x7fff05536010
```

Vetores e Ponteiros - ii

É possível iterar ponteiros:

```
int vetor[10];  
for (int i=0; i<10; i++){  
    cin >> vetor[i];  
}  
int* ptr = vetor;  
for (int i=0; i<10; i++){  
    cout << ptr[i] << endl;  
}
```

Usamos ptr para acessar os elementos de vetor!

Ponteiros e Registros

Quando os ponteiros apontam para registros, o acesso aos campos desses registros pode ser realizado por meio do operador ->:

```
struct Ponto{  
    int X, Y;  
};  
...  
Ponto P1, *P2, *P3;  
cin >> P1.X >> P1.Y;  
P2 = &P1;  
P3 = &P1;  
cout << "P1: X = " << P1.X << " Y = " << P1.Y << endl;  
cout << "P2: X = " << (*P2).X << " Y = " << (*P2).Y << endl;  
cout << "P3: X = " << P3->X << " Y = " << P3->Y << endl;
```



Passagem de Parâmetros - i

Passagem por valor/cópia:

```
int foo1(int a, int b);
```

⇒ As variáveis passadas como parâmetro não tem seu valor modificado dentro da função.

Passagem por referência:

```
int foo2(int &a, int &b);
```

⇒ As variáveis passadas como parâmetro refletem fora da função as alterações que sofrerem dentro dela.

Passagem de Parâmetros - ii

Passagem por ponteiros:

```
int foo3(int *a, int* b);
```

⇒ Esse é um caso específico de passagem por cópia, mas a cópia é do endereço da variável. Assim, os valores também podem ser alterados (como na passagem por referência), uma vez que se tem acesso à variável (por conta de seu endereço). Essa passagem também recebe o nome de *passagem por referência indireta*.

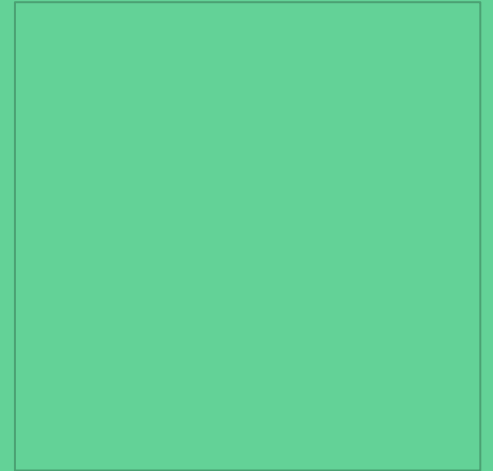
Passagem de Parâmetros - iii

Vários desenvolvedores preferem utilizar passagens por ponteiros que por referência, pelo fato que a passagem por ponteiro é mais legível:

```
int foo2(int &a, int &b);  
int foo3(int *a, int* b);  
int x, y;  
foo2(x,y); // cópia ou referência?  
foo3(&x,&y);
```

A chamada por ponteiros deixa explícito que o valor pode ser alterado pela função.

Alocação dinâmica de memória



Alocação automática de variáveis

A alocação de variáveis implica em reservar um espaço na memória para o armazenamento de seus dados.

Com a alocação automática, o espaço de memória a ser utilizado é gerenciado pelo compilador.

Uma variável é alocada quando é utilizada dentro do escopo e desalocada ao término desse escopo. Esse escopo pode ser um laço ou uma função, por exemplo.

Com a alocação temos pouco controle sobre quando alocar e desalocar.

Alocação dinâmica de variáveis

Em diversas situações, queremos ir além dos limites da alocação automática, reservando grandes espaços de memória ou então determinando que determinado evento exige alocação de mais memória e determinado evento permite a liberação de memória.

Variáveis alocadas dinamicamente não têm nome, elas são utilizadas por meio de seus endereços, ou seja, utilizamos ponteiros para essa tarefa.

O operador new

Em C++ a alocação dinâmica é feita por meio do operador new:

- new recebe o tipo de dado a ser alocado (ou seja, é um **ponteiro tipado**)
- new encontra o bloco de memória com o tamanho necessário
- new retorna o endereço (da primeira posição) do bloco alocado

O endereço de memória retornado pelo operador new usualmente é armazenado em um ponteiro.

Exemplo

```
int *ptr;
```

```
(...)
```

```
ptr = new int;
```

```
*ptr = 13;
```



Nesse exemplo, alocou-se um bloco de memória para armazenar um inteiro e o endereço desta alocação é armazenado no ponteiro ptr.

Desalocando memória

Quando um espaço de memória é alocado utilizando o operador `new`, este espaço não é desalocado automaticamente pelo computador.

O desalocação do espaço de memória alocado com o operador `new` é de responsabilidade do programador.

Para desalocar espaço de memória é utilizado o comando `delete`.

Alocação Dinâmica de Vetores

O operador `new` é geralmente utilizado para alocar pedaços maiores de memória: vetores, strings, estruturas de dados, etc... Para alocar um vetor de `N` posições:

```
int *ptr = new int[N];
```

Os elementos do vetor alocado podem ser acessados assim como os elementos de um vetor comum:

```
ptr[0] = 5;  
ptr[8] = 27;
```

Nesse caso, para desalocar o vetor, é necessário usar `[]` após `delete`:

```
delete[] ptr;
```

Precisamos disso?

E porque usar alocação e não apenas

```
int tam;  
cin >> tam;  
int vetor[tam];  
???
```

Essa técnica é conhecida como VLA:

http://en.wikipedia.org/wiki/Variable-length_array

<http://stackoverflow.com/questions/1887097/variable-length-arrays-in-c>

Limitações do VLA

Além de não ser um padrão (pode não ser suportado em todos os compiladores ou equipamentos), a técnica do VLA é limitada pelo tamanho da stack reservada ao aplicativo.

Apesar de ser útil para casos mais usuais, é bastante limitada.

Forma recomendada é usar `new/delete`.

Alocação Dinâmica de Matrizes

Para alocar matrizes, basta lembrar que uma matriz pode ser enxergada como um vetor de vetores. Nesse caso, alocam-se as linhas e depois as colunas de cada linha:

```
int m, n;  
int **matriz;  
cin >> m >> n;  
matriz = new int*[m];  
for (int i = 0; i < m; i++){  
    matriz[i] = new int[n];  
}
```



Desalocando a matriz

Como a matriz foi alocada em duas operações, é necessário também desalocar em duas operações:

```
for (int i = 0; i < m; i++){  
    // aqui se desalocam as colunas de cada linha  
    delete[] matriz[i];  
}  
  
delete[] matriz; // aqui se desalocam as linhas
```



Retornando vetores - i

Outro motivo para uso de ponteiros é a possibilidade de retornar vetores em funções. Suponha que você quer criar uma função que gera um vetor de tamanho desconhecido N e o retorna.

C++ não permite retornar vetores. Como fazer isto?

Uma possibilidade é passar um vetor como parâmetro e modificá-lo dentro da função. Isso não resolve nosso problema, já que não sabemos o tamanho N , além disso queremos que o vetor seja gerado dentro da função, e não fora.

Solução: Podemos alocar o vetor dinamicamente dentro da função e retornar o seu endereço.

Retornando vetores - ii

```
int* geraVetor(int &n){  
    int *V = new int[n];  
    return V;  
}  
  
int main() {  
    int n;  
    cin >> n;  
    int *meuVetor = geraVetor(n);  
    (...)  
    delete [] meuVetor;  
    return 0;  
}
```



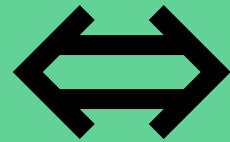
Alocação de Memória em C

C++ é uma evolução do C e trouxe os operadores `new` e `delete`. Em C, a alocação/desalocação geralmente é feita por meio das funções `malloc()` e `free()`:

```
#include <cstdlib>
int n;
int *vetor;
cin >> n;
vetor = (int *) malloc(sizeof(int)*n);
(...)
free(vetor);
```

O ponteiro devolvido por `malloc()`, ao contrário de `new`, não é tipado.

Redimensionamento de vetores



Meu vetor encheu, e agora?

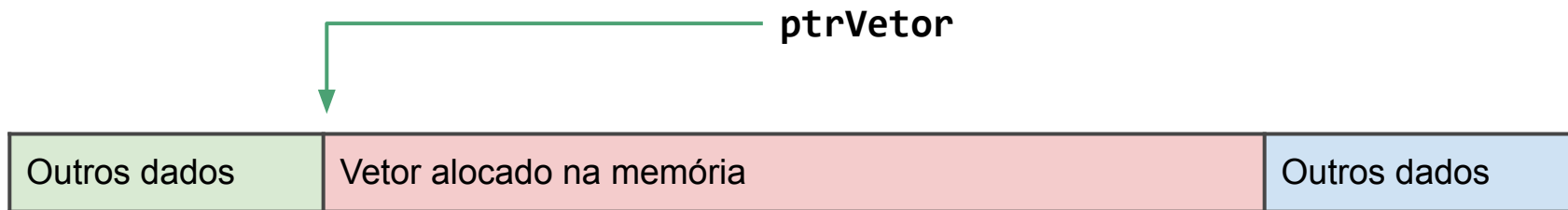
Uma questão que costuma aparecer na solução de diversos problemas utilizando vetores é quando a capacidade inicial não é mais suficiente para resolver o problema.

Suponha que você tenha alocado um vetor com capacidade inicial para 100 elementos, foi enchendo-o aos poucos e agora, após ele cheio, precisa adicionar mais 20 elementos...



Meu vetor encheu, e agora?

Um problema é que com a memória vai sendo utilizada e pode não haver espaço na sequência...



Ou seja, na grande maioria dos casos não é possível expandir o vetor original para suportar a nova capacidade.

Meu vetor encheu, e agora?

A solução para isso envolve alocação dinâmica. E, para que o processo seja mais simples, é necessário que o vetor original também tenha sido alocado dessa forma.

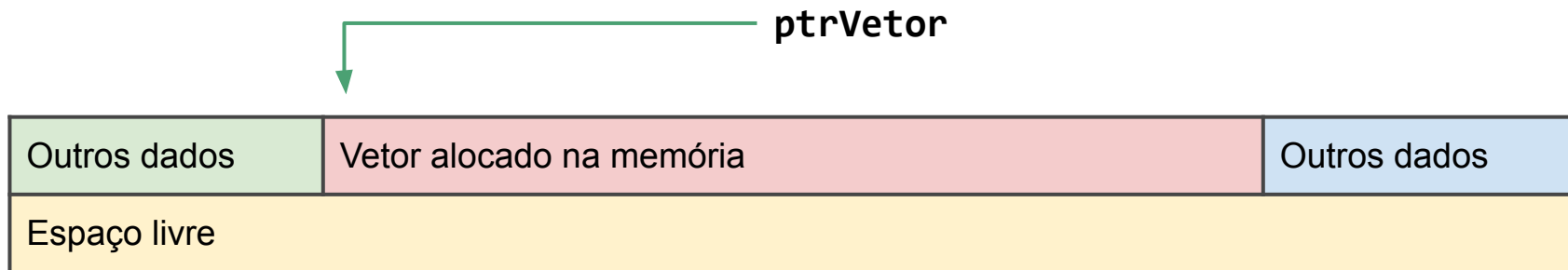
Como vetores alocados dinamicamente são, na prática, ponteiros, é possível mudar o endereço apontado pelo vetor para uma nova região, maior que a anterior...

Isso não é possível com um vetor normal, já que ele é um ponteiro constante (não pode mudar o valor armazenado, o endereço de seus dados).

Como assim, mudar o endereço?

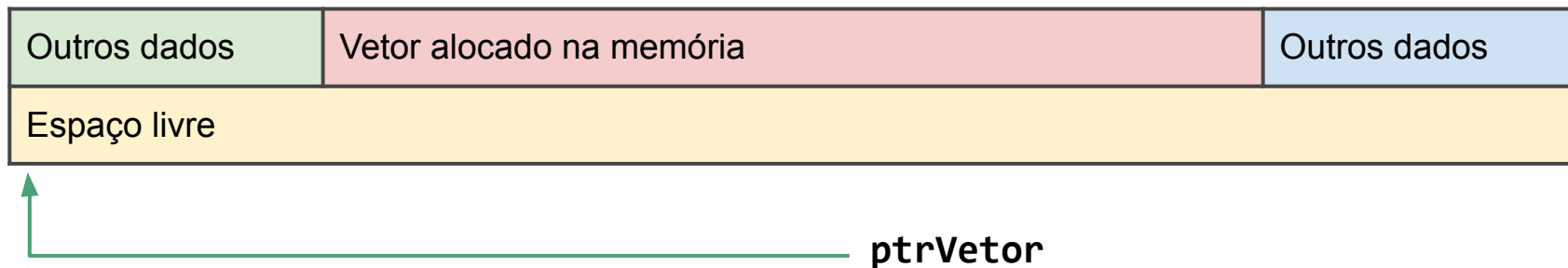
Nós podemos modificar o endereço do vetor alocado para uma região livre da memória que tenha espaço suficiente para comportar a nova capacidade.

Ou seja, mudamos o ponteiro para apontar daqui...



Como assim, mudar o endereço?

Nós podemos modificar o endereço do vetor alocado para uma região livre da memória que tenha espaço suficiente para comportar a nova capacidade.



Para que ele aponte agora para cá...

Entendi, então basta mudar o endereço?

Obviamente isso não basta e nem deve ser a primeira ação a ser feita... Senão, iremos perder todos os dados iniciais...

Precisamos antes de mudar o endereço, alocar a nova região e copiar os dados para esse novo local.



Entendi, então basta mudar o endereço?

Obviamente isso não basta e nem

deve ser assim.
Senão, ***Como assim,
alocar a nova região
e copiar os dados???***

Precisa-se mudar o endereço, alocar a nova região e copiar os dados para esse novo local.



Entendi, então basta mudar o endereço?

Obviamente isso não
deve ser
Senão, *Como a*
dados *alocar o*
Precisa *e copiar*
endereço, alocar a
copiar os dados p
local.

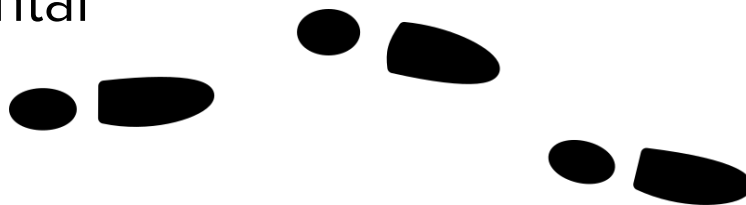
**DON'T
PANIC**



Vamos ao passo a passo

Para redimensionarmos um vetor alocado dinamicamente, seguimos quatro passos:

1. Alocamos um novo vetor, com o tamanho desejado, em uma variável temporária.
2. Copiamos os dados do vetor original para o novo vetor, usando `for`, `memcpy()` ou `copy()`.
3. Desalocamos o vetor original.
4. Fazemos o ponteiro original apontar para o novo local.



Vamos ao passo a passo

Para redimensionarmos um vetor alocado, seguimos quatro passos:

E pronto!

- usando `for`, `memcpy()` ou `copy()`.
- Desalocamos o vetor original.
- Fazemos o ponteiro original apontar para o novo local.



Dá para desenhar?

Suponha o seguinte trecho de código:

```
int* ptrVetor;
```

...

```
ptrVetor = new int[100];
```

...

E suponha que agora você precisa aumentar o espaço alocado para 150 posições nesse vetor.

Dá para desenhar?

Vamos então alocar o novo espaço e copiar os dados:

```
ptrVetor = new int[100];
```

```
...
```

```
int* temp = new int[150];  
for (int i = 0; i < 100; i++)  
    temp[i] = ptrVetor[i];
```

```
...
```

Dá para desenhar?

Agora desalocamos o vetor original e apontamos para a nova região:

```
for (int i = 0; i < 100; i++)  
    temp[i] = ptrVetor[i];
```

...

```
delete[] ptrVetor;  
ptrVetor = temp;
```

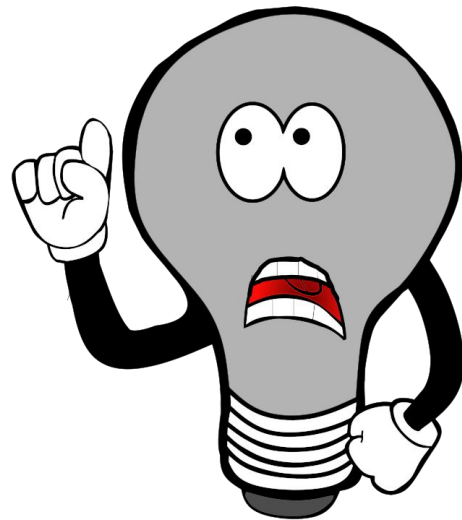
Com isso, ptrVetor vai apontar para o mesmo endereço apontado por temp.

Agora é só fazer `delete[]` em `temp`, certo?

Não!

Se você desalocar a região alocada por `temp`, na verdade, você não está removendo `temp`, estará desalocando o espaço que é agora apontado por `ptrVetor`.

Ou seja, o processo já terminou, não precisa fazer mais nada!



E essa cópia, não é ineficiente?

Será que não há uma forma mais eficiente de copiar esses dados???

...

```
int* temp = new int[150];  
for (int i = 0; i < 100; i++)  
    temp[i] = ptrVetor[i];
```

...

Palma, palma, não priemos cânico!

Caso os dados não possuam atributos dinâmicos, é possível usar `memcpy()`. Essa é uma alternativa muito eficiente, que copia um trecho inteiro de memória de uma região para outra.

Se for um vetor de registros, e esse registro possuir algum campo dinâmico (uma variável do tipo `string`, por exemplo), é possível usar a função `copy()`.



Usando memcpy()

A função `memcpy()`, da biblioteca `<cstring>`, pode ser utilizada para copiar regiões inteiras de memória de uma variável para outra:

```
void* memcpy(void* destino, const void* origem,  
             size_t num_bytes);
```

Nesse caso, `void*` indica ponteiro sem tipo e `size_t` é um inteiro sem sinal, utilizado para tamanhos de tipos, variáveis, etc.

Os dados precisam estar em uma região sequencial da memória, ou seja, não dá para copiar registros que tenham atributos dinâmicos (como uma variável do tipo `string`, por exemplo)

Exemplo - memcpy()

```
#include <cstring>
```

```
(...)
```

```
int vetor1[20] = {0,3,11,61,8,7,-6,2,13,47,  
                  91,12,5,4,-1,12,41,53,9,33};
```

```
int vetor2[10];
```

```
memcpy(vetor2,vetor1,sizeof(int)*10);
```

```
cout << "Após cópia" << endl;
```

```
for (int i=0; i<10; i++)
```

```
    cout << vetor2[i] << endl;
```

```
return 0;
```

Exemplo - memcpy()

```
#include <cstring>
```

```
(...)
```

```
int vetor1[20] =
```

```
int vetor2[10];
```

```
memcpy(vetor2, vet
```

```
cout << "Após cóp
```

```
for (int i=0; i<1
```

```
    cout << vetor
```

```
return 0;
```

```
[joukim@harpia tmp]$ ./a.out
```

```
Após cópia
```

```
0
```

```
3
```

```
11
```

```
61
```

```
8
```

```
7
```

```
-6
```

```
2
```

```
13
```

```
47
```

```
[joukim@harpia tmp]$
```

Exemplo - memcpy()

```
#include <cstring>
```

```
(...)
```

```
int vetor
```

```
int vetor
```

```
memcpy(v
```

```
cout <<
```

```
for (int i=0; i<1
```

```
cout << vetor
```

```
return 0;
```

```
[joukim@harpia tmp]$ ./a.out
```

```
Após cópia
```

Note que, neste exemplo, foram copiados 10 elementos do vetor principal, o objetivo aqui não foi mostrar o redimensionamento, apenas que a cópia também pode ser parcial.

```
-6
```

```
2
```

```
13
```

```
47
```

```
[joukim@harpia tmp]$
```

Usando copy()

A função `copy()`, da biblioteca `<algorithm>`, também pode ser utilizada para copiar regiões inteiras de memória de uma variável para outra:

```
copy (InputIterator first, InputIterator last,  
      OutputIterator result);
```

Nesse caso, `first` é o endereço da primeira posição a ser copiada, `last` é o endereço da última e `result` é o local para onde os dados serão copiados.

Exemplo - copy()

```
#include <algorithm>
```

```
(...)
```

```
int vetor1[20] = {0,3,11,61,8,7,-6,2,13,47,  
                  91,12,5,4,-1,12,41,53,9,33};
```

```
int vetor2[10];
```

```
copy(vetor1,vetor1+10,vetor2);
```

```
cout << "Após cópia" << endl;
```

```
for (int i=0; i<10; i++)
```

```
    cout << vetor2[i] << endl;
```

```
return 0;
```

Exemplo - copy()

```
#include <algorithm>
```

```
(...)
```

```
int vetor1[20] = {0,3,11,91,4,-1,1,5,9,33};
```

```
int vetor2[10];
```

```
copy(vetor1,vetor1+10,vetor2);
```

```
cout << "Após cópia" << endl;
```

```
for (int i=0; i<10; i++)
```

```
    cout << vetor2[i] << endl;
```

```
return 0;
```

Aqui passamos o endereço da última posição a ser copiada

Exemplo - copy()

```
#include <algorithm>
```

```
(...)
```

```
int vetor1[20] = {0,3,11,91,4,-1,11,1,10,9,33};
```

```
int vetor2[10];
```

```
copy(vetor1,&vetor1[9],vetor2);
```

```
cout << "Após cópia" << endl;
```

```
for (int i=0; i<10; i++)
```

```
    cout << vetor2[i] << endl;
```

```
return 0;
```

Também poderíamos
fazer desta forma!

Agora é com você!

Desenvolva aplicativos para redimensionar vetores. Experimente a cópia com `for`, `memcpy()` ou `copy()` e verifique os resultados.



Cuidados com o uso de ponteiros



Erros comuns - i

```
int* ptr;  
*ptr = 10;
```

Armazenar valor em uma posição referenciada por um ponteiro, sem antes ter direcionado o ponteiro para uma região útil de memória (ex.: uma variável).



Erros comuns - ii

```
int* ptr1;  
ptr1 = 10;
```

```
int n = 10;  
int* ptr2 = &n;  
int* ptr3 = *ptr2;
```

Fazer o ponteiro apontar para uma região de memória sobre a qual não se sabe o que possui.

Os dois exemplos ao lado geram o mesmo erro, com relação a `ptr1` e `ptr2`, respectivamente.



Erros comuns - iii

```
int* foo() {  
    int n = 10;  
    int* ptr = &n;  
    return ptr;  
}
```



Outro erro comum é fazer o ponteiro apontar para uma variável que vai deixar de existir antes do ponteiro deixar de existir, fazendo com que se tente acessar uma região indevida de memória (ponteiro solto).

No caso, como `n` é local, ela é desalocada antes do término da função, mas `ptr` continua com seu endereço inicial.

Erros comuns - iv

```
int* ptr1 = new int;  
*ptr1 = 10;  
(...)  
int* ptr2 = ptr1;  
(...)  
delete ptr1;  
(...)  
*ptr2 = 20; // erro!
```

Ponteiros soltos também podem surgir quando apontam para uma variável alocada dinamicamente e já liberada para outros usos, mas o ponteiro continua direcionando para ela.



Erros comuns - v

```
int* ptr = new int;  
*ptr = 10;  
(...)  
int p = 20;  
ptr = &p;
```



Uma variável dinâmica perdida ocorre quando se faz uma alocação, mas o ponteiro é redirecionado para outro local, sem que o espaço alocado seja liberado ou sem direcionar outro ponteiro para a região.

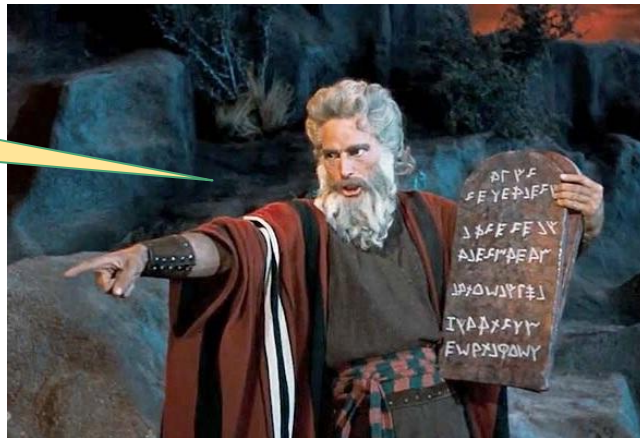
A variável inicialmente alocada por ptr ficou perdida na memória, sem poder ser usada ou liberada.

Recomendações para Uso de Ponteiros - i

Ponteiros devem ser sempre inicializados, seja com valor nulo (NULL ou `nullptr`), seja com o endereço de uma variável:

```
int n = 10;  
int* ptr1 = &n;  
int* ptr2 = NULL;
```

Inicialize!



Recomendações para Uso de Ponteiros - ii

Quando se desaloca uma região de memória referenciada por um ponteiro, deve-se apontá-lo para NULL, quando há possibilidade de reuso do ponteiro:

```
int* ptr = new int;  
*ptr = 10;  
(...)  
delete ptr;  
ptr = NULL;
```

Invalide!

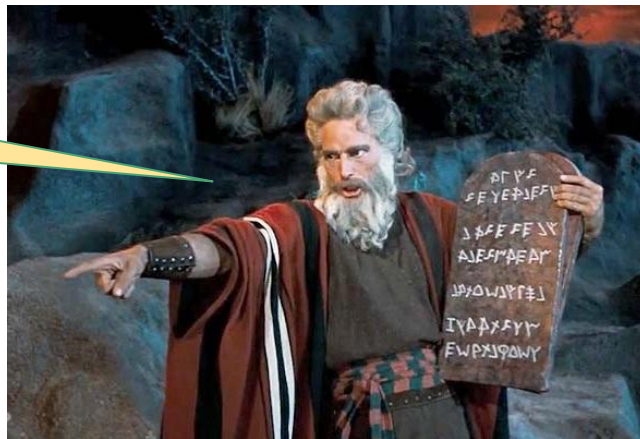


Recomendações para Uso de Ponteiros - iii

Quando não há certeza que o ponteiro aponta para uma região válida, deve-se verificar sua validade, comparando-o com NULL, por exemplo, antes do uso:

```
int* ptr;  
(...)  
if (ptr) {  
    // if (ptr != NULL)  
    *ptr = 10;  
}
```

Teste!

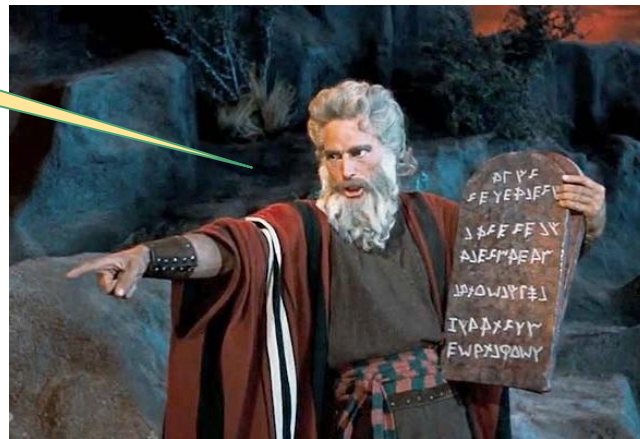


Recomendações para Uso de Ponteiros - iv

Toda memória alocada dinamicamente deve ser desalocada dinamicamente:

```
int* ptr = new int;  
int* vet = new int[10];  
(...)  
delete ptr;  
delete [] vet;
```

Desaloque!



Sobre o Material



Sobre este material

Material produzido coletivamente, principalmente pelos seguintes professores do DCC/UFLA:

- Janderson Rodrigo de Oliveira
- Joaquim Quinteiro Uchôa
- Juliana Galvani Greggi
- Renato Ramos da Silva

Inclui contribuições de outros professores do setor de Fundamentos de Programação do DCC/UFLA.

Esta obra está licenciado com uma Licença [Creative Commons Atribuição 4.0 Internacional](https://creativecommons.org/licenses/by/4.0/).