C++

Básico ao Avançado

Não aponte para mim

Heitor Rodrigues Savegnago

UFABC Rocket Design

2017.3

Joga num cout

Joga num cout

- Vocês já tentaram imprimir um vetor?

Joga num cout

- Vocês já tentaram imprimir um vetor?
- Qual resultado obtiveram?

Joga num cout

- Vocês já tentaram imprimir um vetor?
- Qual resultado obtiveram?
- Vocês tem ideia do que é esse valor?

Joga num cout

- Vocês já tentaram imprimir um vetor?
- Qual resultado obtiveram?
- Vocês tem ideia do que é esse valor?
- Esse valor foi sempre igual?

Joga num cout

- Vocês já tentaram imprimir um vetor?
- Qual resultado obtiveram?
- Vocês tem ideia do que é esse valor?
- Esse valor foi sempre igual?

```
int vetor[10]{0,1,2,3,4,5,6,7,8,9};
//...
std::cout << vetor << std::endl;
```

Joga num cout

- Vocês já tentaram imprimir um vetor?
- Qual resultado obtiveram?
- Vocês tem ideia do que é esse valor?
- Esse valor foi sempre igual?

```
int vetor[10]{0,1,2,3,4,5,6,7,8,9};
//...
std::cout << vetor << std::endl;
```

- Este valor é o endereço onde o vetor está na memória!

Variáveis também tem endereço

Variáveis também tem endereço

- Todas as coisas *declaradas* no programa têm um endereço de memória

Variáveis também tem endereço

- Todas as coisas *declaradas* no programa têm um endereço de memória
- Este endereço será apresentado como um número hexadecimal de 12 (?) dígitos

Variáveis também tem endereço

- Todas as coisas *declaradas* no programa têm um endereço de memória
- Este endereço será apresentado como um número hexadecimal de 12 (?) dígitos
- Há um operador criado para coletar este valor

Variáveis também tem endereço

- Todas as coisas *declaradas* no programa têm um endereço de memória
- Este endereço será apresentado como um número hexadecimal de 12 (?) dígitos
- Há um operador criado para coletar este valor
- Este operador é chamado *referenciador*

Variáveis também tem endereço

- Todas as coisas *declaradas* no programa têm um endereço de memória
- Este endereço será apresentado como um número hexadecimal de 12 (?) dígitos
- Há um operador criado para coletar este valor
- Este operador é chamado *referenciador* ou oprador de endereço

Variáveis também tem endereço

- Todas as coisas *declaradas* no programa têm um endereço de memória
- Este endereço será apresentado como um número hexadecimal de 12 (?) dígitos
- Há um operador criado para coletar este valor
- Este operador é chamado *referenciador* ou oprador de endereço
- Utiliza o *e comercial* (`&`)

Variáveis também tem endereço

- Todas as coisas *declaradas* no programa têm um endereço de memória
- Este endereço será apresentado como um número hexadecimal de 12 (?) dígitos
- Há um operador criado para coletar este valor
- Este operador é chamado *referenciador* ou oprador de endereço
- Utiliza o *e comercial* (`&`)
- Operador unário e de consulta

Variáveis também tem endereço

- Todas as coisas *declaradas* no programa têm um endereço de memória
- Este endereço será apresentado como um número hexadecimal de 12 (?) dígitos
- Há um operador criado para coletar este valor
- Este operador é chamado *referenciador* ou oprador de endereço
- Utiliza o *e comercial* (&)
- Operador unário e de consulta

```
&<nome>;
```

Variáveis também tem endereço

- Todas as coisas *declaradas* no programa têm um endereço de memória
- Este endereço será apresentado como um número hexadecimal de 12 (?) dígitos
- Há um operador criado para coletar este valor
- Este operador é chamado *referenciador* ou oprador de endereço
- Utiliza o *e comercial* (&)
- Operador unário e de consulta

```
&<nome>;
```

- E o exemplo?

Variáveis também tem endereço

- Todas as coisas *declaradas* no programa têm um endereço de memória
- Este endereço será apresentado como um número hexadecimal de 12 (?) dígitos
- Há um operador criado para coletar este valor
- Este operador é chamado *referenciador* ou oprador de endereço
- Utiliza o *e comercial* (&)
- Operador unário e de consulta

```
&<nome>;
```

- E o exemplo? Depois...

Variáveis também tem endereço

- Todas as coisas *declaradas* no programa têm um endereço de memória
- Este endereço será apresentado como um número hexadecimal de 12 (?) dígitos
- Há um operador criado para coletar este valor
- Este operador é chamado *referenciador* ou oprador de endereço
- Utiliza o *e comercial* (&)
- Operador unário e de consulta

```
&<nome>;
```

- E o exemplo? Depois...
- Mas podemos exibir valores!

Dedos na cara

Dedos na cara

- Legal, pra que vou usar isso?

Dedos na cara

- Legal, pra que vou usar isso?
- Com o endereço podemos alterar o valor salvo lá

Dedos na cara

- Legal, pra que vou usar isso?
- Com o endereço podemos alterar o valor salvo lá
- Para isso utilizamos algo que serve para apontar para estes endereços

Dedos na cara

- Legal, pra que vou usar isso?
- Com o endereço podemos alterar o valor salvo lá
- Para isso utilizamos algo que serve para apontar para estes endereços
- Os *ponteiros*

Dedos na cara

- Legal, pra que vou usar isso?
- Com o endereço podemos alterar o valor salvo lá
- Para isso utilizamos algo que serve para apontar para estes endereços
- Os *ponteiros*
- Ponteiros são variáveis especiais expecializada em armazenar endereços

Dedos na cara

- Legal, pra que vou usar isso?
- Com o endereço podemos alterar o valor salvo lá
- Para isso utilizamos algo que serve para apontar para estes endereços
- Os *ponteiros*
- Ponteiros são variáveis especiais expecializada em armazenar endereços
- Apresentam os mesmo tipos que as variáveis

Dedos na cara

- Legal, pra que vou usar isso?
- Com o endereço podemos alterar o valor salvo lá
- Para isso utilizamos algo que serve para apontar para estes endereços
- Os *ponteiros*
- Ponteiros são variáveis especiais expecializada em armazenar endereços
- Apresentam os mesmo tipos que as variáveis (inclusive tipos abstratos)

Problemas com tamanhos

Problemas com tamanhos

- *Espera, você está me dizendo*

Problemas com tamanhos

- *Espera, você está me dizendo que essas variáveis que sempre armazenam endereços*

Problemas com tamanhos

- *Espera, você está me dizendo que essas variáveis que sempre armazenam endereços tem mais de um tipo???*

Problemas com tamanhos

- *Espera, você está me dizendo que essas variáveis que sempre armazenam endereços tem mais de um tipo???*
- *Mas os endereços não deveriam ter sempre o mesmo formato???*

Problemas com tamanhos

- *Espera, você está me dizendo que essas variáveis que sempre armazenam endereços tem mais de um tipo???*
- *Mas os endereços não deveriam ter sempre o mesmo formato???*
- Sim!

Problemas com tamanhos

- *Espera, você está me dizendo que essas variáveis que sempre armazenam endereços tem mais de um tipo???*
- *Mas os endereços não deveriam ter sempre o mesmo formato???*
- Sim! e tem um ótimo motivo

Problemas com tamanhos

- *Espera, você está me dizendo que essas variáveis que sempre armazenam endereços tem mais de um tipo???*
- *Mas os endereços não deveriam ter sempre o mesmo formato???*
- Sim! e tem um ótimo motivo
- Vamos brincar com outro operador rapidamente...

Problemas com tamanhos

- *Espera, você está me dizendo que essas variáveis que sempre armazenam endereços tem mais de um tipo???*
- *Mas os endereços não deveriam ter sempre o mesmo formato???*
- Sim! e tem um ótimo motivo
- Vamos brincar com outro operador rapidamente...
- O `sizeof`, o operador de tamanho

Problemas com tamanhos

- *Espera, você está me dizendo que essas variáveis que sempre armazenam endereços tem mais de um tipo???*
- *Mas os endereços não deveriam ter sempre o mesmo formato???*
- Sim! e tem um ótimo motivo
- Vamos brincar com outro operador rapidamente...
- O `sizeof`, o operador de tamanho

```
sizeof(<nome>);
```

Problemas com tamanhos

- *Espera, você está me dizendo que essas variáveis que sempre armazenam endereços tem mais de um tipo???*
- *Mas os endereços não deveriam ter sempre o mesmo formato???*
- Sim! e tem um ótimo motivo
- Vamos brincar com outro operador rapidamente...
- O `sizeof`, o operador de tamanho

```
sizeof(<nome>);
```

- *O que ele faz?*

Problemas com tamanhos

- *Espera, você está me dizendo que essas variáveis que sempre armazenam endereços tem mais de um tipo???*
- *Mas os endereços não deveriam ter sempre o mesmo formato???*
- Sim! e tem um ótimo motivo
- Vamos brincar com outro operador rapidamente...
- O `sizeof`, o operador de tamanho

```
sizeof(<nome>);
```

- *O que ele faz?*
- *O que ele come?*

Problemas com tamanhos

- *Espera, você está me dizendo que essas variáveis que sempre armazenam endereços tem mais de um tipo???*
- *Mas os endereços não deveriam ter sempre o mesmo formato???*
- Sim! e tem um ótimo motivo
- Vamos brincar com outro operador rapidamente...
- O `sizeof`, o operador de tamanho

```
sizeof(<nome>);
```

- *O que ele faz?*
- *O que ele come?*
- *Como se reproduz?*

Problemas com tamanhos

- *Espera, você está me dizendo que essas variáveis que sempre armazenam endereços tem mais de um tipo???*
- *Mas os endereços não deveriam ter sempre o mesmo formato???*
- Sim! e tem um ótimo motivo
- Vamos brincar com outro operador rapidamente...
- O `sizeof`, o operador de tamanho

```
sizeof(<nome>);
```

- *O que ele faz?*
- *O que ele come?*
- *Como se reproduz?*

- Ele retorna a quantidade de bytes que a entidade no argumento ocupa

Problemas com tamanhos

- *Espera, você está me dizendo que essas variáveis que sempre armazenam endereços tem mais de um tipo???*
- *Mas os endereços não deveriam ter sempre o mesmo formato???*
- Sim! e tem um ótimo motivo
- Vamos brincar com outro operador rapidamente...
- O `sizeof`, o operador de tamanho

```
sizeof(<nome>);
```

- *O que ele faz?*
- *O que ele come?*
- *Como se reproduz?*

- Ele retorna a quantidade de bytes que a entidade no argumento ocupa
- Vamos fazer um teste...

Voltando aos ponteiros

Voltando aos ponteiros

- Legal, com esse operador há uma explicação

Voltando aos ponteiros

- Legal, com esse operador há uma explicação
- Cada tipo tem um tamanho característico (ou tem outras diferenças)

Voltando aos ponteiros

- Legal, com esse operador há uma explicação
- Cada tipo tem um tamanho característico (ou tem outras diferenças)
- Esse tamanho mostra quantos bytes uma variável representa na memória

Voltando aos ponteiros

- Legal, com esse operador há uma explicação
- Cada tipo tem um tamanho característico (ou tem outras diferenças)
- Esse tamanho mostra quantos bytes uma variável representa na memória
- Se utilizarmos um ponteiro genérico, não tem como garantir a quantidade correta de bytes alterados no acesso

Voltando aos ponteiros

- Legal, com esse operador há uma explicação
- Cada tipo tem um tamanho característico (ou tem outras diferenças)
- Esse tamanho mostra quantos bytes uma variável representa na memória
- Se utilizarmos um ponteiro genérico, não tem como garantir a quantidade correta de bytes alterados no acesso
- Legal, faz sentido, mas duas dúvidas:

Voltando aos ponteiros

- Legal, com esse operador há uma explicação
- Cada tipo tem um tamanho característico (ou tem outras diferenças)
- Esse tamanho mostra quantos bytes uma variável representa na memória
- Se utilizarmos um ponteiro genérico, não tem como garantir a quantidade correta de bytes alterados no acesso
- Legal, faz sentido, mas duas dúvidas:
  - Como faz?

Voltando aos ponteiros

- Legal, com esse operador há uma explicação
- Cada tipo tem um tamanho característico (ou tem outras diferenças)
- Esse tamanho mostra quantos bytes uma variável representa na memória
- Se utilizarmos um ponteiro genérico, não tem como garantir a quantidade correta de bytes alterados no acesso
- Legal, faz sentido, mas duas dúvidas:
  - Como faz?
  - Acesso. . . ?

Voltando aos ponteiros

- Legal, com esse operador há uma explicação
- Cada tipo tem um tamanho característico (ou tem outras diferenças)
- Esse tamanho mostra quantos bytes uma variável representa na memória
- Se utilizarmos um ponteiro genérico, não tem como garantir a quantidade correta de bytes alterados no acesso
- Legal, faz sentido, mas duas dúvidas:
  - Como faz?
  - Acesso...?
- Vamos começar com a declaração

Declarando

Declarando

- A declaração é quase idêntica à de uma variável comum

Declarando

- A declaração é quase idêntica à de uma variável comum
- Ela ganha uma estrelinha!

Declarando

- A declaração é quase idêntica à de uma variável comum
- Ela ganha uma estrelinha! Mentira, é só um asterisco $*$

Declarando

- A declaração é quase idêntica à de uma variável comum
- Ela ganha uma estrelinha! Mentira, é só um asterisco $*$
- O asterisco está lá só nos ponteiros

Declarando

- A declaração é quase idêntica à de uma variável comum
- Ela ganha uma estrelinha! Mentira, é só um asterisco *
- O asterisco está lá só nos ponteiros

```
<modificadores> <tipo> *<nome>;
```

Declarando

- A declaração é quase idêntica à de uma variável comum
- Ela ganha uma estrelinha! Mentira, é só um asterisco *
- O asterisco está lá só nos ponteiros

```
<modificadores> <tipo> *<nome>;
```

- *Legal, e o tal acesso?*

Declarando

- A declaração é quase idêntica à de uma variável comum
- Ela ganha uma estrelinha! Mentira, é só um asterisco $*$
- O asterisco está lá só nos ponteiros

```
<modificadores> <tipo> *<nome>;
```

- *Legal, e o tal acesso?*
- Precisamos inicializar ainda...

Inicializando

Inicializando

- Mesma ideia que uma variável normal

Inicializando

- Mesma ideia que uma variável normal
- Recebe endereços

Inicializando

- Mesma ideia que uma variável normal
- Recebe endereços
- Pode receber o valor de outro ponteiro

Inicializando

- Mesma ideia que uma variável normal
- Recebe endereços
- Pode receber o valor de outro ponteiro
- Pode apontar pra qualquer coisa

Inicializando

- Mesma ideia que uma variável normal
- Recebe endereços
- Pode receber o valor de outro ponteiro
- Pode apontar pra qualquer coisa Mas nem todas são simples acessar...

Inicializando

- Mesma ideia que uma variável normal
- Recebe endereços
- Pode receber o valor de outro ponteiro
- Pode apontar pra qualquer coisa Mas nem todas são simples acessar...

```
<modificadores> <tipo> <var>;
//...
<modificadores> <tipo> *<nome>(&<var>);
```

Uns exemplos

Uns exemplos

```
bool falso(0);                  //Booleano
char letraA('A');               //Caractere
int cont(1), *Pcont(&cont);     //Inteiro, e ponteiro para
    inteiro
float dados[50]{};              //Vetor de flutuantes
float *Pdados(dados);           //Ponteiro para vetor
char *PletraA(&letraA);         //Apontando para a variável
    lateraA
bool *Pfalso(&falso);           //Apontando para a variável falso
int *P2cont(Pcont);             //Apontando para o mesmo que o
    ponteiro Pcont
```

Outro operador

Outro operador

- Agora chegamos na parte de acesso

Outro operador

- Agora chegamos na parte de acesso
- Não adianta muito saber o endereço se não souber *mandar cartas*

Outro operador

- Agora chegamos na parte de acesso
- Não adianta muito saber o endereço se não souber *mandar cartas*
- Existe um operador para acessar através do endereço, utilizando os ponteiros

Outro operador

- Agora chegamos na parte de acesso
- Não adianta muito saber o endereço se não souber *mandar cartas*
- Existe um operador para acessar através do endereço, utilizando os ponteiros
- Este operador é chamado *derreferenciador*

Outro operador

- Agora chegamos na parte de acesso
- Não adianta muito saber o endereço se não souber *mandar cartas*
- Existe um operador para acessar através do endereço, utilizando os ponteiros
- Este operador é chamado *derreferenciador* ou operador de acesso

Outro operador

- Agora chegamos na parte de acesso
- Não adianta muito saber o endereço se não souber *mandar cartas*
- Existe um operador para acessar através do endereço, utilizando os ponteiros
- Este operador é chamado *derreferenciador* ou operador de acesso
- Utiliza o asterisco (*)

Outro operador

- Agora chegamos na parte de acesso
- Não adianta muito saber o endereço se não souber *mandar cartas*
- Existe um operador para acessar através do endereço, utilizando os ponteiros
- Este operador é chamado *derreferenciador* ou operador de acesso
- Utiliza o asterisco (*)
- É um operador unário e de consulta

Outro operador

- Agora chegamos na parte de acesso
- Não adianta muito saber o endereço se não souber *mandar cartas*
- Existe um operador para acessar através do endereço, utilizando os ponteiros
- Este operador é chamado *derreferenciador* ou operador de acesso
- Utiliza o asterisco (∗)
- É um operador unário e de consulta, mas...

Outro operador

- Agora chegamos na parte de acesso
- Não adianta muito saber o endereço se não souber *mandar cartas*
- Existe um operador para acessar através do endereço, utilizando os ponteiros
- Este operador é chamado *derreferenciador* ou operador de acesso
- Utiliza o asterisco (*)
- É um operador unário e de consulta, mas...
- Ele é de consulta para o valor do ponteiro, mas com ele se altera a o valor apontado

Outro operador

- Agora chegamos na parte de acesso
- Não adianta muito saber o endereço se não souber *mandar cartas*
- Existe um operador para acessar através do endereço, utilizando os ponteiros
- Este operador é chamado *derreferenciador* ou operador de acesso
- Utiliza o asterisco (*)
- É um operador unário e de consulta, mas...
- Ele é de consulta para o valor do ponteiro, mas com ele se altera a o valor apontado

```
*<nome>;
```

Legal, quando usar?

Legal, quando usar?

```
//...
void MDC(int *primeiro, int *segundo) //Parâmetros são
    ponteiros
  while(*segundo!=0)
  {
    int resto(*primeiro%*segundo);
    *primeiro=*segundo;                //A memória é alterada
    *segundo=resto;
  }
}
//...
int main()
{
  int alpha(234), beta(5493);
  MDC(&alpha, &beta);          //O endereço é o argumento
  int C(alpha);                //C vale 3
  return 0;
}
```

Uma setinha $\rightarrow$

Uma setinha $\rightarrow$

- Temos um caso especial de ponteiro

Uma setinha $\rightarrow$

- Temos um caso especial de ponteiro
- Quando apontamos para um tipo abstrato acontece um problema com precedência...

Uma setinha $\rightarrow$

- Temos um caso especial de ponteiro
- Quando apontamos para um tipo abstrato acontece um problema com precedência...
- A precedência do operador de acesso é menor que a do operador de membro (.)

Uma setinha $\rightarrow$

- Temos um caso especial de ponteiro
- Quando apontamos para um tipo abstrato acontece um problema com precedência...
- A precedência do operador de acesso é menor que a do operador de membro (.)
- Podemos utilizar parênteses...

Uma setinha →

- Temos um caso especial de ponteiro
- Quando apontamos para um tipo abstrato acontece um problema com precedência...
- A precedência do operador de acesso é menor que a do operador de membro (.)
- Podemos utilizar parênteses...

```
//...
(*<nome>).<membro>;    //Caso de variável membro
(*<nome>).<membro>(); //Caso de função membro
//...
```

Uma setinha →

- Temos um caso especial de ponteiro
- Quando apontamos para um tipo abstrato acontece um problema com precedência...
- A precedência do operador de acesso é menor que a do operador de membro (.)
- Podemos utilizar parênteses...

```
//...
(*<nome>).<membro>;    //Caso de variável membro
(*<nome>).<membro>(); //Caso de função membro
//...
```

- Tá ruim...

Uma setinha →

- Temos um caso especial de ponteiro
- Quando apontamos para um tipo abstrato acontece um problema com precedência...
- A precedência do operador de acesso é menor que a do operador de membro (.)
- Podemos utilizar parênteses...

```
//...
(*<nome>).<membro>;    //Caso de variável membro
(*<nome>).<membro>(); //Caso de função membro
//...
```

- Tá ruim...
- E se existir um operador pra facilitar isso?

Uma setinha →

- Temos um caso especial de ponteiro
- Quando apontamos para um tipo abstrato acontece um problema com precedência...
- A precedência do operador de acesso é menor que a do operador de membro (.)
- Podemos utilizar parênteses...

```
//...
(*<nome>).<membro>;    //Caso de variável membro
(*<nome>).<membro>(); //Caso de função membro
//...
```

- Tá ruim...
- E se existir um operador pra facilitar isso?
- Existe!

Uma setinha →

- Temos um caso especial de ponteiro
- Quando apontamos para um tipo abstrato acontece um problema com precedência...
- A precedência do operador de acesso é menor que a do operador de membro (.)
- Podemos utilizar parênteses...

```
//...
(*<nome>).<membro>;    //Caso de variável membro
(*<nome>).<membro>(); //Caso de função membro
//...
```

- Tá ruim...
- E se existir um operador pra facilitar isso?
- Existe! E é uma seta

Uma setinha →

- Temos um caso especial de ponteiro
- Quando apontamos para um tipo abstrato acontece um problema com precedência...
- A precedência do operador de acesso é menor que a do operador de membro (.)
- Podemos utilizar parênteses...

```
//...
(*<nome>).<membro>;    //Caso de variável membro
(*<nome>).<membro>(); //Caso de função membro
//...
```

- Tá ruim...
- E se existir um operador pra facilitar isso?
- Existe! E é uma seta
- Aliás, quase uma seta (->)

Quase uma seta. . .

Quase uma seta...

```
//...
<nome>-><membro>;    //Caso de variável membro
<nome>-><membro>(); //Caso de função membro
//...
```

Quase uma seta...

```
//...
<nome>-><membro>;    //Caso de variável membro
<nome>-><membro>(); //Caso de função membro
//...
```

- Utilizamos este operador para acessar membros quando temos apenas o ponteiro

Quase uma seta...

```
//...
<nome>-><membro>;    //Caso de variável membro
<nome>-><membro>(); //Caso de função membro
//...
```

- Utilizamos este operador para acessar membros quando temos apenas o ponteiro
- Ele acessa o endereço apontado e, lá, acessa o membro

Quase uma seta...

```
//...
<nome>-><membro>;    //Caso de variável membro
<nome>-><membro>(); //Caso de função membro
//...
```

- Utilizamos este operador para acessar membros quando temos apenas o ponteiro
- Ele acessa o endereço apontado e, lá, acessa o membro
- Muito

Quase uma seta...

```
//...
<nome>-><membro>;    //Caso de variável membro
<nome>-><membro>(); //Caso de função membro
//...
```

- Utilizamos este operador para acessar membros quando temos apenas o ponteiro
- Ele acessa o endereço apontado e, lá, acessa o membro
- Muito, muito

Quase uma seta...

```
//...
<nome>-><membro>;    //Caso de variável membro
<nome>-><membro>(); //Caso de função membro
//...
```

- Utilizamos este operador para acessar membros quando temos apenas o ponteiro
- Ele acessa o endereço apontado e, lá, acessa o membro
- Muito, muito, muitíssimo

Quase uma seta...

```
//...
<nome>-><membro>;    //Caso de variável membro
<nome>-><membro>(); //Caso de função membro
//...
```

- Utilizamos este operador para acessar membros quando temos apenas o ponteiro
- Ele acessa o endereço apontado e, lá, acessa o membro
- Muito, muito, muitíssimo, muito mesmo utilizado em esturutras de dados

Um caso de uso

Um caso de uso

```
struct meuTipo
{
  int valor1, valor2;
};
//...
int somarMt(meuTipo *S)
{
  return S->valor1 + S->valor2;
}
//...
int main()
{
  meuTipo mem;
  int soma;
  mem.valor1 = 10;
  mem.valor2 = 20;
  //...                  //Atribui valor ao vetor
  soma = somarMt(&mem);
  //...
  return 0;
}
```

Continhas

Continhas

- Por que essa carinha triste?

Continhas

- Por que essa carinha triste? ☹

Continhas

- Por que essa carinha triste? ☹
- Podemos ser melhores do que isso...

Continhas

- Por que essa carinha triste? ☹
- Podemos ser melhores do que isso...
- Mas vocês precisam saber que isso axiste

Continhas

- Por que essa carinha triste? ☹
- Podemos ser melhores do que isso...
- Mas vocês precisam saber que isso axiste
- Operadores ++ e −−

Continhas

- Por que essa carinha triste? ☹
- Podemos ser melhores do que isso...
- Mas vocês precisam saber que isso axiste
- Operadores ++ e −− (Soma e subtração também)

Continhas

- Por que essa carinha triste? ☹
- Podemos ser melhores do que isso...
- Mas vocês precisam saber que isso axiste
- Operadores ++ e −− (Soma e subtração também)
- Caminhar para frente e para trás

Continhas

- Por que essa carinha triste? ☹
- Podemos ser melhores do que isso...
- Mas vocês precisam saber que isso axiste
- Operadores ++ e −− (Soma e subtração também)
- Caminhar para frente e para trás
- Problema:

Continhas

- Por que essa carinha triste? ☹
- Podemos ser melhores do que isso...
- Mas vocês precisam saber que isso axiste
- Operadores ++ e −− (Soma e subtração também)
- Caminhar para frente e para trás
- Problema: Perder o valor original

Continhas

- Por que essa carinha triste? ☹
- Podemos ser melhores do que isso...
- Mas vocês precisam saber que isso axiste
- Operadores ++ e −− (Soma e subtração também)
- Caminhar para frente e para trás
- Problema: Perder o valor original
- *E daí?*

Continhas

- Por que essa carinha triste? ☹
- Podemos ser melhores do que isso...
- Mas vocês precisam saber que isso axiste
- Operadores ++ e −− (Soma e subtração também)
- Caminhar para frente e para trás
- Problema: Perder o valor original
- *E daí?*
- Vocês entederão isso na próxima aula

Continhas

- Por que essa carinha triste? ☹
- Podemos ser melhores do que isso...
- Mas vocês precisam saber que isso axiste
- Operadores ++ e −− (Soma e subtração também)
- Caminhar para frente e para trás
- Problema: Perder o valor original
- *E daí?*
- Vocês entederão isso na próxima aula
- Tem a ver com

Continhas

- Por que essa carinha triste? ☹
- Podemos ser melhores do que isso...
- Mas vocês precisam saber que isso axiste
- Operadores ++ e −− (Soma e subtração também)
- Caminhar para frente e para trás
- Problema: Perder o valor original
- *E daí?*
- Vocês entederão isso na próxima aula
- Tem a ver com **A**

Continhas

- Por que essa carinha triste? ☹
- Podemos ser melhores do que isso...
- Mas vocês precisam saber que isso axiste
- Operadores ++ e −− (Soma e subtração também)
- Caminhar para frente e para trás
- Problema: Perder o valor original
- *E daí?*
- Vocês entederão isso na próxima aula
- Tem a ver com **A COISA**

Continhas

- Por que essa carinha triste? ☹
- Podemos ser melhores do que isso...
- Mas vocês precisam saber que isso axiste
- Operadores ++ e −− (Soma e subtração também)
- Caminhar para frente e para trás
- Problema: Perder o valor original
- *E daí?*
- Vocês entederão isso na próxima aula
- Tem a ver com **A COISA MAIS**

Continhas

- Por que essa carinha triste? ☹
- Podemos ser melhores do que isso...
- Mas vocês precisam saber que isso axiste
- Operadores ++ e −− (Soma e subtração também)
- Caminhar para frente e para trás
- Problema: Perder o valor original
- *E daí?*
- Vocês entederão isso na próxima aula
- Tem a ver com **A COISA MAIS DIFÍCIL**

Continhas

- Por que essa carinha triste? ☹
- Podemos ser melhores do que isso...
- Mas vocês precisam saber que isso axiste
- Operadores ++ e −− (Soma e subtração também)
- Caminhar para frente e para trás
- Problema: Perder o valor original
- *E daí?*
- Vocês entederão isso na próxima aula
- Tem a ver com **A COISA MAIS DIFÍCIL** (que não é difícil)

Sente o drama

Sente o drama

```
float mediaN2(int N, float *V)
{
  float acc = 0;
  //Dê atenção aos argumentos do for a seguir
  for(float *aux(V + N); V<aux; V++) acc += *V;
  return acc/float(N);
}
//...
#define SIZE 50
int main()
{
  float vetor[SIZE]{};
  float media;
  //...                  //Atribui valor ao vetor
  media = mediaN2(SIZE, vetor);
  //...
  return 0;
}
```

Mais que continhas

Mais que continhas

- Indexado!

Mais que continhas

- Indexado!

```
float mediaN1(int N, float *V)
{
  float acc = 0;
  for(int i = 0; i<N; i++) acc += V[i];
  return acc/float(N);
}
//...
#define SIZE 50
int main()
{
  float vetor[SIZE]{};
  float media;
  //...                  //Atribui valor ao vetor
  media = mediaN1(SIZE, vetor);
  //...
  return 0;
}
```

Vamos testar!