

Registros

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

A grayscale photograph of a hand holding a pen, checking off a list of items on a document. The document has a series of checkboxes, some of which are already marked with an 'X'. The word 'Roteiro' is overlaid in large, bold, black letters.

Roteiro

- Contextualização
- Registros
- Registros e arranjos
- Registros e subprogramas
- Registros e memória

Contextualização



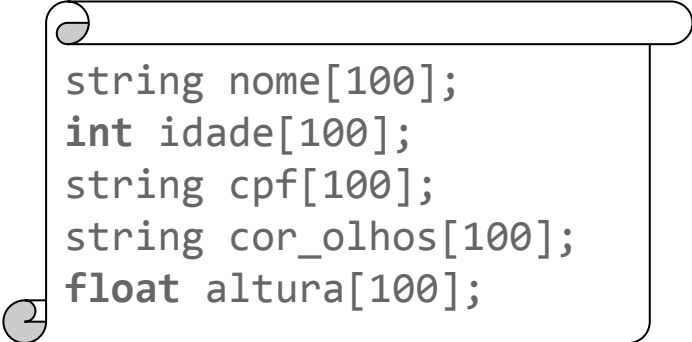
Contextualização

- Como dados de diferentes tipos, porém relacionados semanticamente a uma mesma entidade do mundo real, devem ser armazenados em um programa de computador?
- Por exemplo: cada pessoa possui um nome, um número de CPF, características físicas como cor dos olhos, altura, idade, entre outras informações que a definem de maneira única.
- A questão é: como armazenar de forma adequada informações variadas de tipos diferentes que se referem a uma única entidade (como estas que descrevem uma pessoa, por exemplo)?
- Poderíamos usar variáveis diferentes para cada informação. Mas, o que fazer caso seja necessário armazenar dados de várias pessoas?

Contextualização

Solução possível com uso de arranjos: cada tipo de dado seria armazenado em um arranjo diferente e o índice seria o ponto de relação entre os dados de uma única pessoa armazenados nos diferentes arranjos, ou seja, o índice *i* de cada arranjo armazena as informações referentes a uma pessoa específica.

Por exemplo, se tivéssemos 100 pessoas:



```
string nome[100];  
int idade[100];  
string cpf[100];  
string cor_olhos[100];  
float altura[100];
```

Neste exemplo, o índice **0** de todos os arranjos denota as informações da primeira pessoa, o índice **1** dos arranjos as informações da segunda pessoa, e assim por diante.

Contextualização

Esta abordagem possui uma **desvantagem**: um erro na indexação de apenas um dos arranjos seria suficiente para tornar todo o conjunto de dados inconsistente. Por exemplo, imagine que em um dado trecho de programa fizéssemos a seguinte leitura para um certo indivíduo:

```
cin >> nome[i];  
cin >> idade[i];  
cin >> cpf[i];  
cin >> cor_olhos[j];  
cin >> altura[i];
```

Neste exemplo, o programador (por desatenção ou outro motivo qualquer) escreveu erroneamente uma outra variável como índice do vetor **cor_olhos**, tornando assim todo o conjunto inconsistente.

Contextualização

- Solução desejada
 - Para o problema discutido até então, uma solução mais adequada seria se uma única posição de um dado arranjo fosse capaz de armazenar simultaneamente todas as informações referentes a um único indivíduo.
 - Uma vez que arranjos são estruturas compostas **homogêneas**, ou seja, armazenam apenas dados de um mesmo tipo, faz-se necessário então um outro tipo de estrutura.

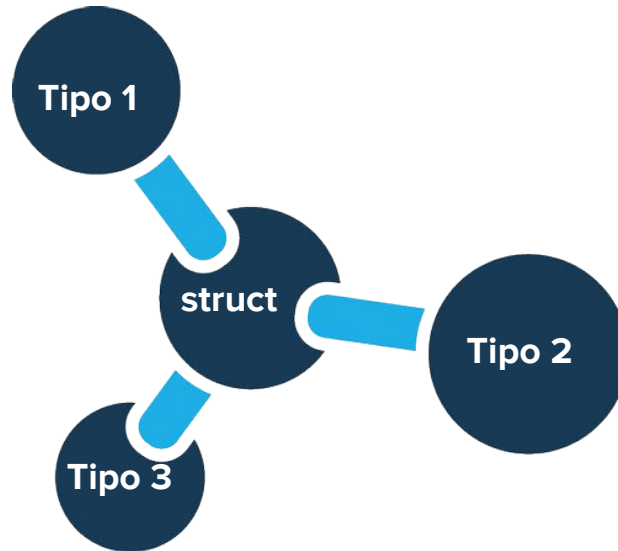


Registros



Registros

- **Definição:** registros (struct) são estruturas compostas heterogêneas, ou seja, capazes de armazenar dados de tipos diferentes agrupados em um mesmo elemento de dados.



Registros

- Sintaxe em C++ (definição da estrutura):

```
struct identificador_estrutura{  
    tipo_dado_1 identificador_campo_1;  
    tipo_dado_2 identificador_campo_2;  
    //..  
    tipo_dado_N identificador_campo_N;  
};
```

Cada registro pode definir inúmeros **campos** (tantos quanto necessário). Cada campo expressa um determinado tipo de informação que se deseja armazenar através da estrutura.

Registros

- Sintaxe em C++ (definição da estrutura):

```
struct identificador_estrutura{  
    tipo_dado_1 identificador_campo_1;  
    tipo_dado_2 identificador_campo_2;  
    //..  
    tipo_dado_N identificador_campo_N;  
};
```

; é obrigatório

Cada registro pode definir inúmeros **campos** (tantos quanto necessário). Cada campo expressa um determinado tipo de informação que se deseja armazenar através da estrutura.

Registros

- Considere o exemplo anterior, no qual precisávamos criar uma estrutura para armazenar as informações do nome, idade, número do CPF, altura e cor dos olhos de um dado indivíduo. Assim teríamos:

```
struct pessoa{  
    string nome;  
    int idade;  
    string cpf;  
    float altura;  
    string cor_olhos;  
};
```

Neste momento, estamos apenas definindo as características (campos) que a estrutura **pessoa** deve possuir.

Caso tenhamos que realmente ler as informações de uma dada pessoa, é necessário declararmos uma variável do tipo da estrutura heterogênea criada e só então seremos capazes de manipular a variável.

Registros

- **Sintaxe em C++ (declaração de variável):**

```
identificador_estrutura identificador_variável;
```

- No nosso exemplo, poderíamos fazer em C++:

```
pessoa individuo;
```

Neste exemplo, **pessoa** é o tipo de dado da variável (neste caso, o tipo é definido pelo próprio programador) e **individuo** é o nome da variável daquele tipo. Não se esqueça que, qualquer informação fornecida, será armazenada e manipulada na variável (individuo).

Registros

- **Sintaxe em C (declaração de variável):**

```
struct identificador_estrutura identificador_variável;
```

- No nosso exemplo, poderíamos fazer em C:

```
struct pessoa individuo;
```



Em C, **não** colocar a palavra reservada **struct** antes do **identificador_estrutura** gerará um erro de sintaxe.

Registros

- Assim como qualquer variável composta, é necessário que a partir de uma variável do tipo registro possamos manipular seus campos isoladamente. Sendo assim...
- **Sintaxe em C/C++ (acesso aos campos de um registro):**

```
identificador_variável.identificador_campo;
```

Registros

- Assim como qualquer variável composta, é necessário que a partir de uma variável do tipo registro possamos manipular seus campos isoladamente. Sendo assim...
- Sintaxe em C/C++ (acesso aos campos de um registro):**

```
identificador_variável.identificador_campo;
```

O ponto é usado como separador da variável para o seu campo.

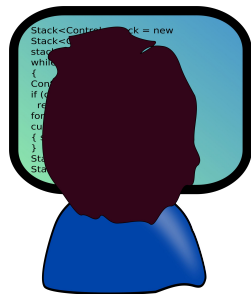
Registros - Exemplo

Construir um programa que armazene em um registro as informações de uma dada pessoa. Seu programa deve ler e exibir as informações fornecidas pelo usuário.

As informações a serem consideradas serão:

- Nome;
- Idade;
- CPF;
- Altura;
- Cor dos olhos.

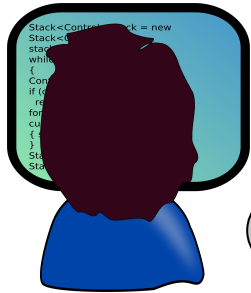




```
#include <iostream>
using namespace std;
struct pessoa{
    string nome;
    int idade;
    string cpf;
    float altura;
    string cor_olhos;
};
int main(){
    pessoa individuo;
    cin >> individuo.nome;
    cin >> individuo.idade;
    cin >> individuo.cpf;
    cin >> individuo.altura;
    cin >> individuo.cor_olhos;
```

A definição do registro geralmente é feita fora de qualquer subprograma, inclusive da função **main()**, para permitir declaração de variáveis desse novo tipo em qualquer função ou procedimento.

Declaração de uma variável chamada **individuo** do tipo **pessoa**. A variável possui todos os campos especificados na definição do registro.



●

```
cout << "Nome: " << individuo.nome << endl;  
cout << "Idade: " << individuo.idade << endl;  
cout << "CPF: " << individuo.cpf << endl;  
cout << "Altura: " << individuo.altura << endl;  
cout << "Cor dos olhos: " << individuo.cor_olhos << endl;  
return 0;
```

Deve-se sempre colocar o nome da variável antes do identificador de campo de interesse do registro.

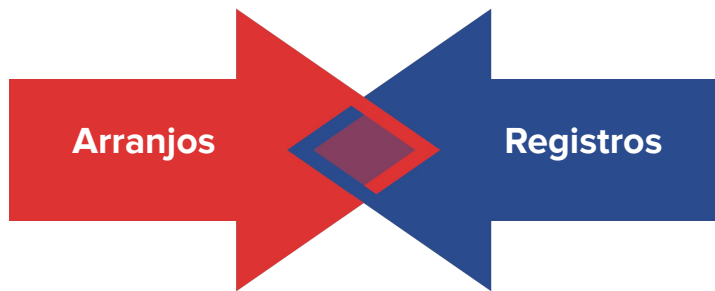


Registros e arranjos



Registros e arranjos

- Os conceitos de estruturas de dados compostas homogêneas (arranjos) e heterogêneas (registros) podem ser combinados.



- Com isso, é possível criar:
 - Arranjos (vetores ou matrizes) cujos elementos são registros.
 - Registros cujos campos são arranjos (vetores ou matrizes).

Registros e arranjos - Exemplo

- Problema: você precisa fazer o levantamento de livros disponíveis em uma livraria. Faça um programa em que todos os livros serão armazenados com título, autor, quantidade em estoque e preço.
- Seu programa deve ser capaz, ainda, de realizar buscas pelo título do livro e mostrar o preço e a quantidade disponível em estoque do livro procurado.
- Considere que a livraria possui ao todo 500 livros.

```
#include <iostream>
using namespace std;
struct livro {
    string titulo, autor;
    int quantidade;
    float preco;
};
int main() {
    const int acervo = 500;
    int i, posicao;
    string busca;
    livro livraria[acervo];
    for (i = 0; i < acervo; i++) {
        getline(cin, livraria[i].titulo);
        getline(cin, livraria[i].autor);
        cin >> livraria[i].quantidade >> livraria[i].preco;
        cin.ignore();
    }
```

Definição do registro

Variáveis:

acervo: quantidade de livros

i: variável auxiliar para repetições

busca: título do livro da busca

livraria: vetor de dados dos livros

posicao: índice do livro buscado no vetor de dados. Caso o livro não esteja presente no conjunto, assume-se o valor -1.

Exemplo: Livraria

1/2

```
#include <iostream>
using namespace std;
struct livro {
    string titulo, autor;
    int quantidade;
    float preco;
};
int main() {
    const int acervo = 500;
    int i, posicao;
    string busca;
    livro livraria[acervo];
    for (i = 0; i < acervo; i++) {
        getline(cin, livraria[i].titulo);
        getline(cin, livraria[i].autor);
        cin >> livraria[i].quantidade >> livraria[i].preco;
        cin.ignore();
    }
```

cin.ignore()
evita a
leitura da
quebra de
linha pelo
getline()

Lembrando... como o título do livro tem espaços em seu nome, usamos `cin.ignore()` antes da próxima leitura usando `getline()`

```
#include <iostream>
using namespace std;
struct livro {
    string titulo, autor;
    int quantidade;
    float preco;
};
int main() {
    const int acervo = 500;
    int i, posicao;
    string busca;
    livro livraria[acervo];
    for (i = 0; i < acervo; i++) {
        getline(cin, livraria[i].titulo);
        getline(cin, livraria[i].autor);
        cin >> livraria[i].quantidade >> livraria[i].preco;
        cin.ignore();
    }
```



```
cout << "Título do livro de busca: ";  
getline(cin,busca);
```

```
i = 0;
```

```
posicao = -1;
```

```
while ((i < acervo) and (busca != livraria[i].titulo)) i++;
```

```
if (busca == livraria[i].titulo) posicao = i;
```

```
if (posicao == -1) cout << "O livro não está disponível." << endl;
```

```
else {
```

```
    cout << "Preço: R$" << livraria[i].preco << endl;
```

```
    cout << "Quantidade em estoque: " << livraria[i].quantidade << endl;
```

```
}
```

```
}
```



```
cout << "Título do livro de busca: ";  
getline(cin, busca);
```

*Buscando o
elemento
procurado
no vetor*

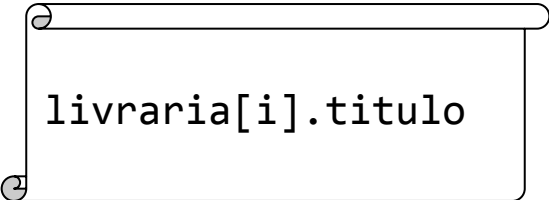
```
i = 0;  
posicao = -1;  
while ((i < acervo) and (busca != livraria[i].titulo)) i++;  
if (busca == livraria[i].titulo) posicao = i;  
  
if (posicao == -1) cout << "O livro não está disponível." << endl;  
else {  
    cout << "Preço: R$" << livraria[i].preco << endl;  
    cout << "Quantidade em estoque: " << livraria[i].quantidade << endl;  
}  
}
```

Registros e arranjos

É importante observar que no caso de variáveis que sejam **vetores de registros**, para podermos manipular as informações armazenadas nos campos de cada posição do vetor, devemos seguir a **sintaxe**:

`identificador_variável_vetor[índice_interesse].identificador_campo`

Exemplo:



```
livraria[i].titulo
```

Registros e arranjos - Exemplo

- A título de exemplificação de vetores como campos, considere que ao invés de uma única livraria, você estivesse trabalhando com uma rede com 5 unidades diferentes.
- Cada unidade possui exatamente os mesmos livros, com os mesmos preços. Contudo, em cada uma, a quantidade de exemplares disponíveis em estoque podem ser diferentes.
- Neste caso, poderíamos declarar o campo **quantidade** do registro como um vetor de cinco posições (uma posição para cada unidade).

Registros e arranjos - Exemplo

- Exemplo (continuação):
 - Ideia geral de um campo de registro do tipo vetor:

```
struct livro {  
    string titulo, autor;  
    int quantidade[5];  
    float preco;  
};
```



Tente modificar por completo o programa do exemplo anterior, de modo que ele funcione para o caso das 5 unidades da rede da livraria.

Registros e arranjos - Exemplo

- Exemplo (continuação):
 - Para lermos nosso vetor de dados, poderíamos então fazer:

Repetição para o
vetor de livros

Repetição para o vetor de
quantidades de cada livro

```
for (i = 0; i < acervo; i++) {  
    getline(cin, livraria[i].titulo);  
    getline(cin, livraria[i].autor);  
    for (j = 0; j < 5; j++) {  
        cin >> livraria[i].quantidade[j];  
    }  
    cin >> livraria[i].preco;  
    cin.ignore();  
}
```

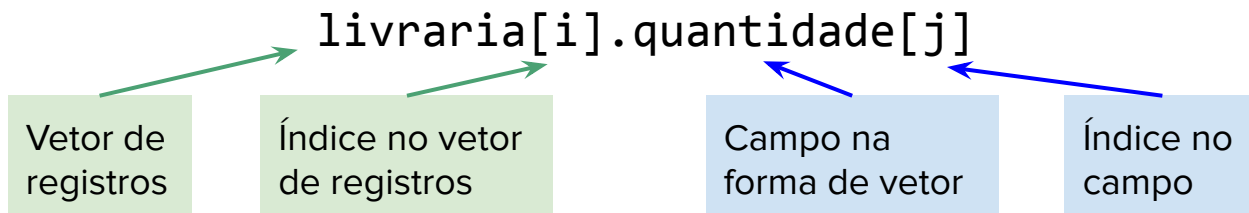


Registros e arranjos

No caso de variáveis que sejam **registros cujos campos são vetores**, para podermos manipular as informações armazenadas nos índices de um determinado campo, devemos seguir a **sintaxe**:

```
identificador_variável_registro.identificador_campo_vetor[índice_interesse]
```

Como no nosso exemplo anterior, estávamos trabalhando com um vetor de registros (arranjo externo de livros), que possuíam um campo do tipo vetor (arranjo interno de quantidades), tivemos que fazer:



Atribuição de registros

Suponha que seja necessário atribuir o valor de um registro a outro.

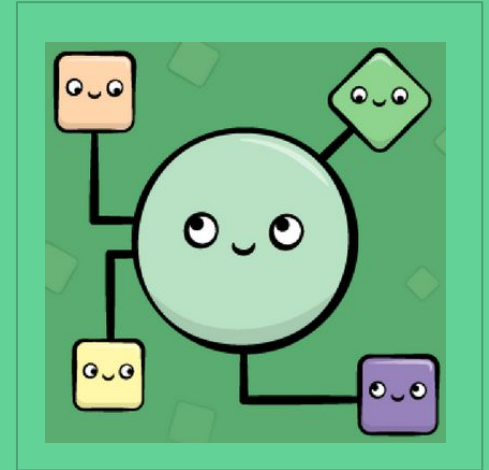
Não é necessário atribuir cada campo do registro, um por um. É possível atribuir o registro inteiro de uma única vez.

Atribuição de registros - Exemplo

```
struct pessoa {  
    string nome;  
    int idade;  
    string cpf;  
    float altura;  
    string olhos;  
};
```

```
pessoa x, y, aux;  
...  
...  
// trocando os valores de x e y  
aux = x;  
x = y;  
y = aux;
```

Registros e subprogramas



Registros e subprogramas

Em C++, além de campos (valores), registros também podem ter funções e procedimentos. Nesse tipo de situação em programação, é comum denominar os campos como **atributos** e as funções e procedimentos como **métodos**.

Essa técnica é base para o paradigma de programação de orientação a objetos (OO). Nesta disciplina, o uso de subprogramas dentro de registros **não é solicitado** e aparece apenas para mostrar que dados e processamento podem ser agrupados, posto que OO é o paradigma de programação dominante.

Registros e subprogramas

No caso de variáveis do tipo **registro** que possuem **subprogramas** (procedimentos e funções) em suas definições, devemos seguir a **sintaxe** abaixo para podermos realizar as chamadas destes subprogramas:

```
identificador_variável_registro.identificador_subprograma(listas_parâmetros)
```

Ou seja, o método é acessado de maneira similar a um atributo, mas com a sintaxe de subprograma.

Registros e subprogramas - Exemplo

- Problema: faça um programa que exiba a norma ou módulo de um ponto.
- Seu programa deve:
 - Definir um registro que representa a entidade ponto;
 - Ler as coordenadas de 10 pontos diferentes e exibir o módulo de cada um.

Registros e subprogramas - Exemplo

- Problema: faça um programa que exiba a norma ou módulo de um ponto.
- Seu

O módulo de um ponto é sua distância até a origem (0,0). Ou seja, dadas as coordenadas x e y de um ponto P, seu módulo é dado por

$$|P| = \sqrt{x^2 + y^2}$$

 - módulo de
 - cada um.

Registros e subprogramas - Exemplo

Exemplo: Módulo de um ponto

1/2

```
#include <iostream>
#include <math.h>
using namespace std;
struct ponto{
    int X, Y;
    void atribuir(int A, int B){
        X = A;
        Y = B;
    }
    float modulo( ){
        return sqrt(pow(X, 2) + pow(Y, 2));
    }
};
```



Registros e subprogramas - Exemplo

Exemplo: Módulo de um ponto

1/2

```
#include <iostream>
#include <math.h>
using namespace std;
struct ponto{
    int X, Y;
    void atribuir(int A, int B){
        X = A;
        Y = B;
    }
    float modulo( ){
        return sqrt(pow(X, 2) + pow(Y, 2));
    }
};
```

Exemplo de
procedimento




Exemplo de
função

Registros e subprogramas - Exemplo

Exemplo: Módulo de um ponto

2/2



```
int main() {  
    int A, B;  
    ponto P;  
  
    for (int i = 0; i < 10; i++) {  
        cin >> A >> B;  
        P.atribuir(A, B);  
        cout << "Módulo: " << P.modulo( ) << endl;  
    }  
    return 0;  
}
```



Registros e subprogramas - Exemplo

Exemplo: Módulo de um ponto

2/2

```
int main() {  
    int A, B;  
    ponto P;  
  
    for (int i = 0; i < 10; i++) {  
        cin >> A >> B;  
        P.atribuir(A, B);  
        cout << "Módulo: " << P.modulo( ) << endl;  
    }  
    return 0;  
}
```

Leitura das
coordenadas
de cada ponto



Chamadas dos
subprogramas
definidos dentro
do registro

Isso tá me cheirando outra coisa...

O uso de procedimentos em registros faz com que esse tipo de dado compartilhe várias semelhanças com classes, um dos elementos centrais em programação orientada a objetos (POO).



Em C++, inclusive, classes (class) e registros (struct) possuem várias semelhanças, diferenciando-se apenas em alguns detalhes. Podemos pensar, de uma maneira mais abrangente, que classes são registros que possuem procedimentos, além dos atributos...

Como assim?

Vamos pegar um exemplo, o tipo string. O tipo string é, na verdade, uma classe e ***poderia ser implementada*** de uma maneira similar à seguinte:

```
class basic_string {  
    char* texto;  
    unsigned tamanho;  
    unsigned capacidade;  
  
    int size(); // retorna o tamanho utilizado pela string  
  
    //etc...  
};
```

Como assim?

Vamos pegar um exemplo, o tipo string. O tipo string é, na verdade, uma classe e ***poderia ser implementada*** de uma maneira similar à seguinte:

```
class basic_string {  
    char* texto;  
    unsigned tamanho;  
    unsigned capacidade;
```

Vetor de caracteres alocado dinamicamente
(veremos mais à frente como isso é feito, por enquanto apenas tenha em mente que esse é um vetor que é criado quando a variável string é criada).

```
    int size(); // retorna o tamanho utilizado pela string
```

```
    //etc...
```

```
};
```

Como assim?

Vamos pegar um exemplo, o tipo string. O tipo string é, na verdade, uma classe e ***poderia ser implementada*** de uma maneira similar à seguinte:

```
class basic_string {  
    char* texto;  
    unsigned tamanho;  
    unsigned capacidade;  
  
    int size(); // retorna o  
  
    //etc...  
};
```

O atributo **tamanho** informa quantas posições do vetor estão ocupadas. Já o atributo **capacidade** informa quantas posições foram alocadas e estão disponíveis para uso no total.

Caso a variável receba uma string de tamanho maior que a capacidade, então um novo vetor (texto) com maior capacidade é alocado e os dados são movidos para esse novo vetor.

Como assim?

Vamos pegar um exemplo, o tipo string. O tipo string é, na verdade, uma classe e ***poderia ser implementada*** de uma maneira similar à seguinte:

```
class basic_string {  
    char* texto;  
    unsigned tamanho;  
    unsigned capacidade;
```

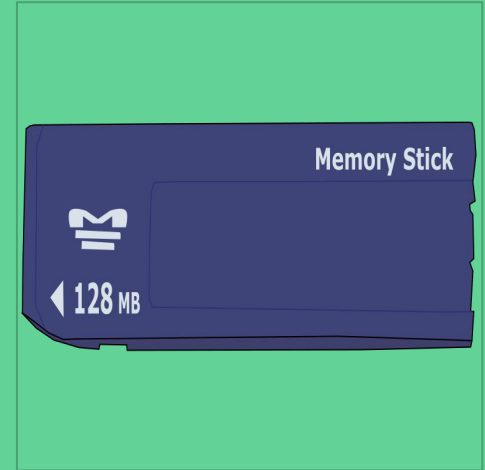
Quando em um código se faz algo do tipo
`i <= palavra.size()`
é uma função interna à classe que está sendo chamada.

```
int size(); // retorna o tamanho utilizado pela string
```

```
//etc...
```

```
};
```


Registros e memória

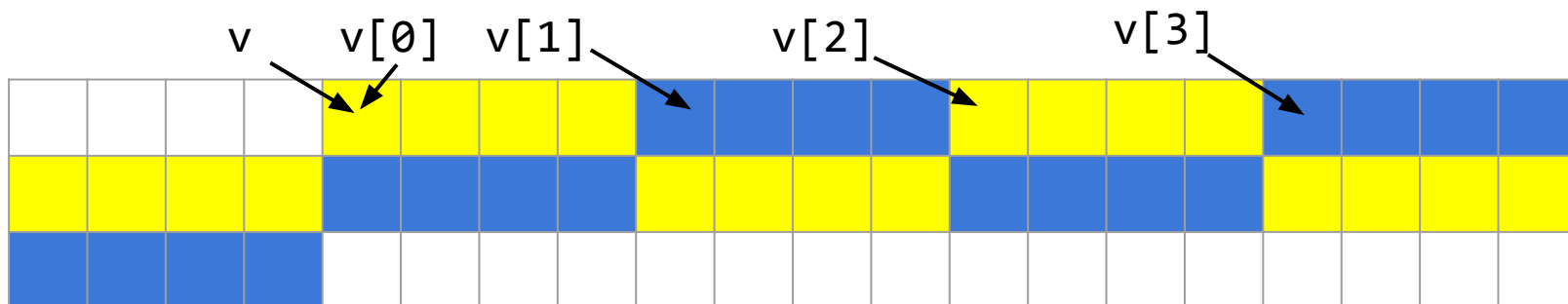


Relembrando vetores e memória

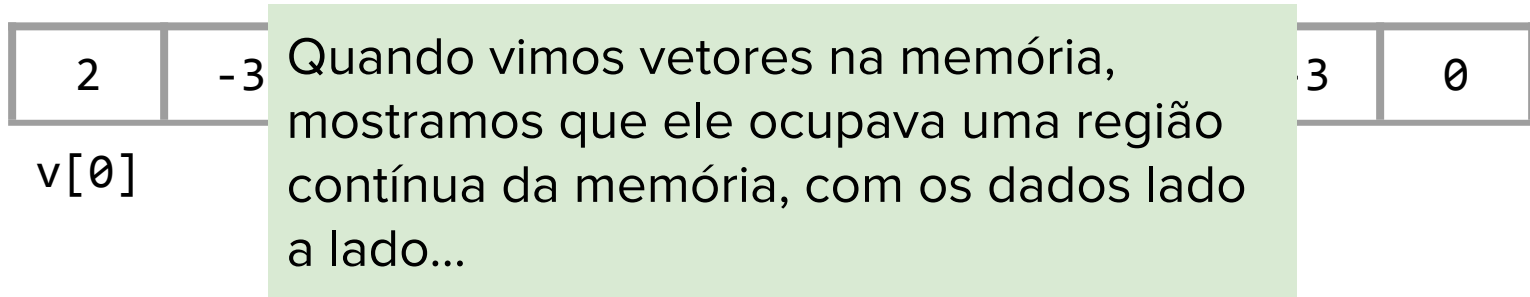
2	-3	101	104	0	1	2	3	-3	0
---	----	-----	-----	---	---	---	---	----	---

v[0]

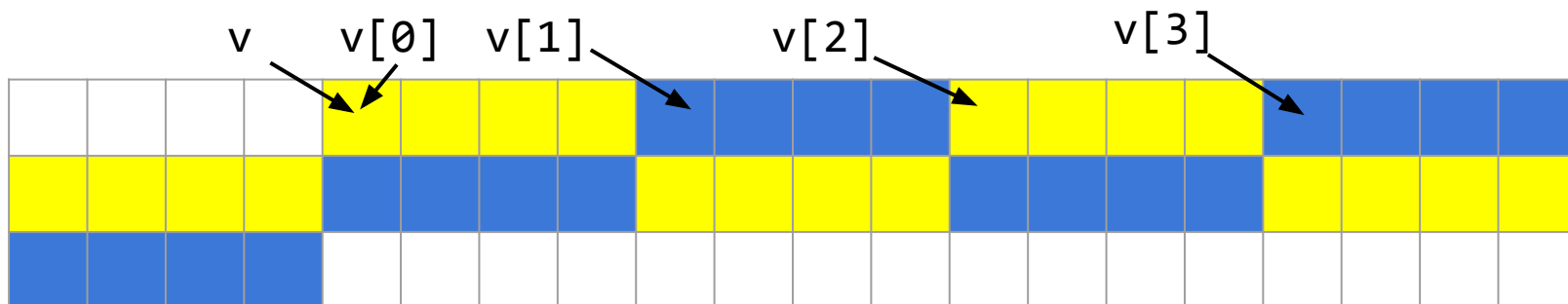
Em memória (considerando 4 bytes por inteiro):



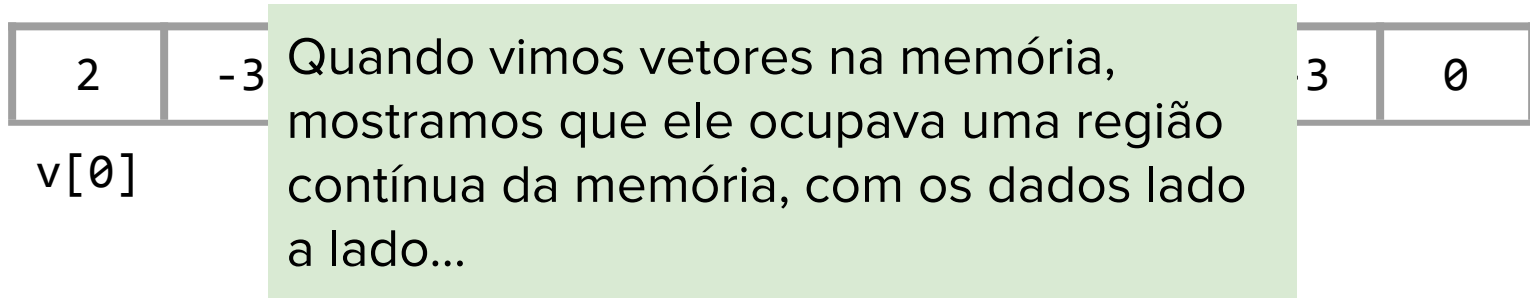
Relembrando vetores e memória



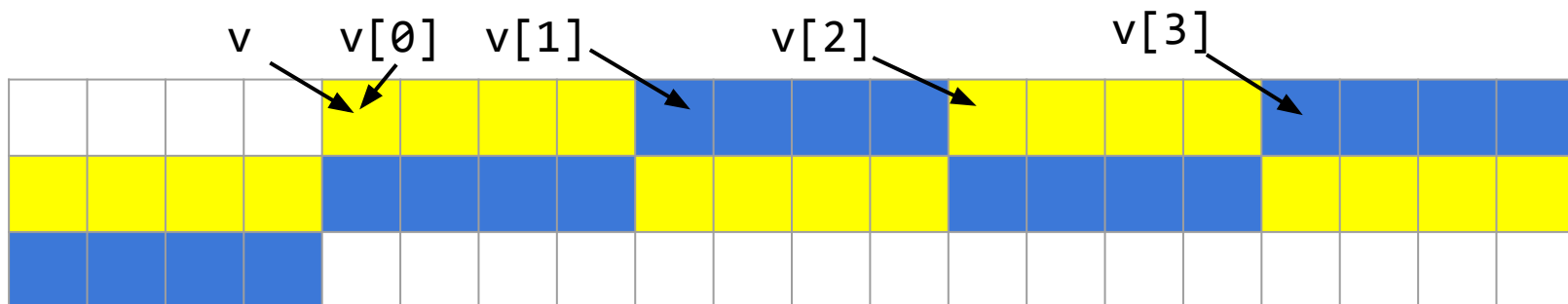
Em memória (considerando 4 bytes por inteiro):



Relembrando vetores e memória



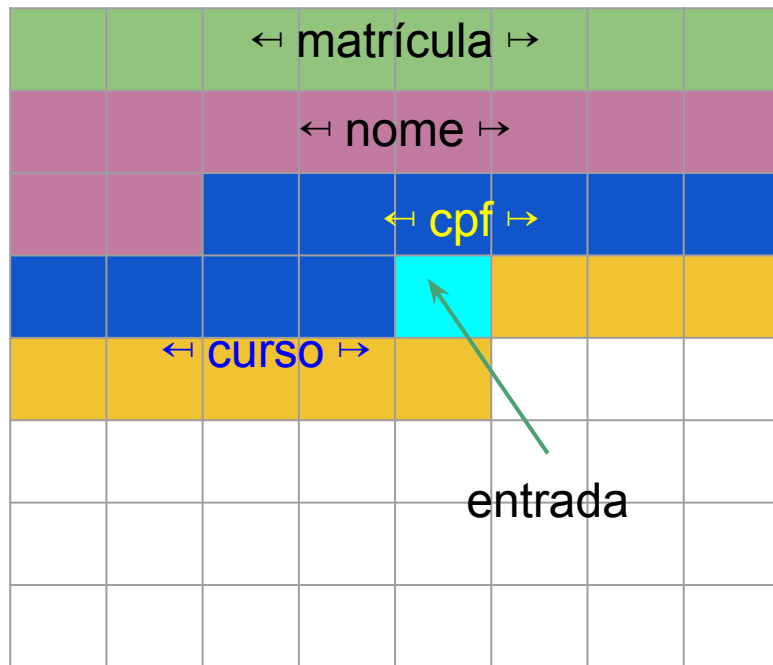
Em memória *E com registros? Será que ocorre a mesma coisa?*



Registros e memória

Registros também são armazenados em uma posição sequencial da memória. Ex:

```
struct aluno {  
    long matricula;  
    char nome [10];  
    char cpf[10];  
    char entrada;  
    long curso;  
};  
aluno jose;
```



Porém, entretanto, contudo, todavia...

Imagine a seguinte situação

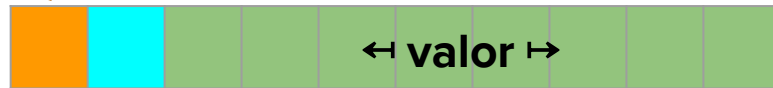
```
struct exemplo {  
    char tipo;  
    char opcao;  
    long valor;  
};
```

Porém, entretanto, contudo, todavia...

Imagine a seguinte situação

```
struct exemplo {  
    char tipo;  
    char opcao;  
    long valor;  
};
```

tipo



opção

*Esta seria uma possível
representação*

Porém, entretanto, contudo, todavia...

Imagine a seguinte situação

```
struct exemplo {  
    char tipo;  
    char opcao;  
    long valor;  
};
```

tipo



*Esta seria uma possível
representação*

**Mas essa representação em geral
não é boa pro hardware**

Não, como assim?

Sistemas computacionais modernos geralmente são projetados para operarem com pelo menos quatro ou oito bytes. Em geral, quando falamos que um sistema é de 32 ou 64 bits, estamos justamente dizendo isso, que ele trabalha com blocos de dados em 32 bits (4 bytes) ou 64 bits (8 bytes).

Então, dá para imaginar que pegar menos que essa quantidade prejudica a eficiência do sistema.

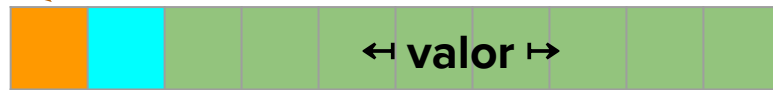


Porém, entretanto, contudo, todavia...

Imagine a seguinte situação

```
struct exemplo {  
    char tipo;  
    char opcao;  
    long valor;  
};
```

tipo



*O problema aqui não são so campos **tipo** e **opção**, já que são apenas 1 byte cada...*

opção

Porém, entretanto, contudo, todavia...

Imagine a seguinte situação

```
struct exemplo {  
    char tipo;  
    char opcao;  
    long valor;  
};
```

O problema aqui não é tipo e opção, já que são apenas 1 byte cada...



Leitura 1

Leitura 2

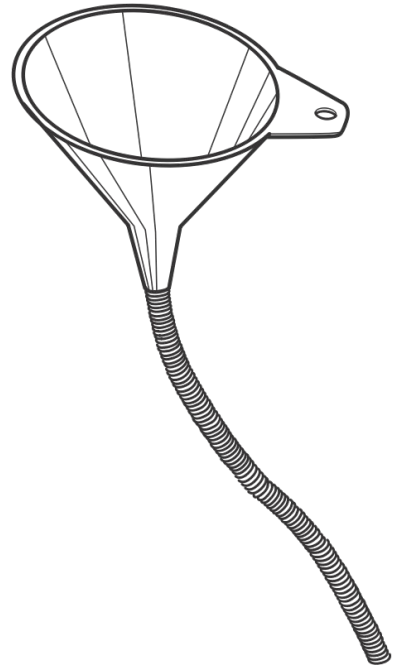
Leitura 3

Se o sistema ler de 4 em 4 bytes, ele terá que fazer 3 leituras para ler o campo **valor**, quando 2 apenas deveriam ser necessárias...

A solução é padding

Assim, para resolver esse problema e aumentar a eficiência, os compiladores podem inserir espaços vazios entre os campos da estrutura para tornar a leitura dos campos/atributos mais eficientes.

Essa técnica recebe o nome de padding.

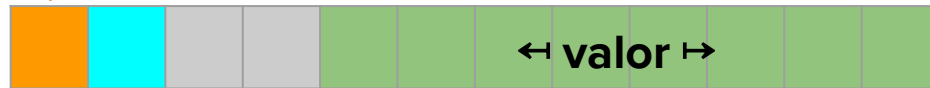


Agora sim!

Imagine a seguinte situação

```
struct exemplo {  
    char tipo;  
    char opcao;  
    long valor;  
};
```

tipo



opção

*Essa é uma representação
mais eficiente e possível*

Agora sim!

Imagine a seguinte situação

```
struct exemplo {  
    char tipo;  
    char opcao;  
    long valor;  
};
```

tipo



opção



Essa é uma representação mais eficiente e possível

Agora apenas 2 leituras são necessária para ler campo **valor**.

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/).