Algoritmos e Estruturas de Dados

(Aula 5 - Criando uma biblioteca de manipulação de vetores)

Prof. Me. Diogo Tavares da Silva
contato: *diogotavares@unibarretos.com.br*

FaculdadeBarretos
www.faculdadebarretos.com.br

Nesta aula aprenderemos...

- Criar tipos de dados mais complexos a partir de tipos mais simples.
- Aprender a definir novos tipos de dados em C/C++
- Aprender como manipular e acessar tipos estruturados
- Compreender os problemas de alocação contígua quando estamos lidando com volumes maiores de dados

Contextualização

- Vamos imaginar que estamos construindo um sistema que cadastra os alunos de uma academia
  - que informações devemos guardar?
    - `char nome[150]`
    - `int idade`
    - `float peso`
    - `float altura`
    - `int id`
  - **Como podemos armazenar essas informações para cada aluno de maneira organizada?**

Tipos estruturados

- Em C/C++, usamos o elemento **struct** (registro) para projetar e manipular **tipos estruturados heterogêneos**.
- Desse modo podemos agrupar todas as variáveis relativas a um elemento lógico em uma única estrutura de acesso e manipulação.
  - análogo ao conceito de atributos de uma classe (programação orientada a objetos)

ex:

- Para o exemplo da academia:

```
struct aluno{
   char nome[150];
   int idade;
   int id;
   float peso;
   float altura;
};
```

ex:

- A partir da estrutura criada podemos declarar um novo tipo de dado, usando o comando **typedef:**

```
typedef struct aluno Aluno;
```

Tipos estruturados

- Desta forma, de maneira análoga, utilizando **struct** e **typedef**, podemos criar tipos mais elaborados
  - A partir de qualquer tipo de dado e estrutura mais simples como variáveis, vetores ou matrizes dos tipos int, float, double, char, ponteiros de tipos, etc.
  - até mesmo outros tipos estruturados

Como acessar os dados dos registros?

- A partir de seus campos
- Para variáveis:
  - nomeDaVariavelDeReg.nomeDoCampo
  - ex:

```
Aluno aln;

aln.idade = 32;

cin >> aln.nome;

cout << aln.peso << endl;

//scanf("%[^\n]s",aln.nome);

//printf("%f",aln.peso);
```

Como acessar os registros

- Para vetores:
  - nomeDoPontDoVetor[indice].nomeDoCampo
  - ex:

```
Aluno aln[30]; //ou alocado dinamicamente

aln[i].idade = 32;

cin >> aln[i].nome;

cout << aln[i].peso;

//scanf(“%[^\n]s”,aln[i].nome);

//printf(“%f”,aln[i].peso);
```

Como acessar os registros

- Para ponteiros:
  - (*nomeDoPonteiro).nomeDoCampo
  - ex:

```
Aluno aln;

ALUNO* pAlu;

pAlu = &aln; //ponteiro recebe end da var

(*pAlu).idade = 32;

cin >> (*pAlu).nome);

cout << (*pAlu).peso) << endl;
```

Como acessar os registros

- Note que para ponteiros sempre devemos deixar o ***nomePonteiro** entre parenteses
  - Por que?
    - Porque o operador “conteúdo de” “*” tem menor precedência que o “acesso ao campo” “.”
  - `(*nomeDoPonteiro).nomeDoCampo`
  - Por essa razão foi criado um operador de acesso próprio para ponteiros: “->”

```
nomeDoPonteiro->nomeDoCampo;
```

Como acessar os registros

- Desse modo, para ponteiros:
  - `(*nomeDoPonteiro).nomeDoCampo`

  OU `nomeDoPonteiro -> nomeDoCampo`
  - ex:

```
Aluno aln;
ALUNO* pAlu;
pAlu = &aln; //ponteiro recebe end da var
pAlu->idade = 32;
cin >> pAlu->nome;
cout << pAlu->peso << endl;
```

Mão na massa

- A partir da declaração de tipo de aluno da academia, como realizado anteriormente, vamos criar um programa que cadastra e edita os dados de alunos:

Passo 1: declaração da estrutura e criação do novo tipo Aluno

```
struct aluno{
   char nome[150];
   int idade;
   int id;
   float peso;
   float altura;
};
typedef struct aluno Aluno;
```

Passo 2: declaração e criação da estrutura dinâmica para armazenar os alunos

```
...funções acima...
main(){
   int n; //número máximo de alunos
   int id=0; //índice do vetor de Alunos
   Aluno* alunos; //ponteiro para o vetor de
alunos que será criado

   puts("Digite o número máximo de alunos");
   cin >> n;
   alunos = criaTurma(n);
//...continuação da main...
}
```

Passo 2: declaração e criação da estrutura dinâmica para armazenar os alunos

```
//função criaTurma()
Aluno* criaTurma(int nAlu){
   Aluno* vAlunos; //vetor de alunos
   vAlunos = new Aluno[nAlu];
   return vAlunos;
}
```

Mão na massa - SUA VEZ!

- Crie as seguintes funcionalidades para o programa:
  - **`Cadastro de alunos`**
  - **`edição de cadastro selecionado`**
  - **`impressão completa dos registros de alunos`**
  - **`lógica de acesso na main para todas as funcionalidades`**

Problemas da Alocação Contínua para registros

- Como discutimos anteriormente, a alocação de espaços contínuos, tanto estática quanto dinâmica, possui o problema de ser necessário um espaço contínuo de memória para armazenar a estrutura.
- Com inteiros ocupando 4 bytes apenas e vetores de no máximo 100 elementos não parece grande coisa
  - 400 bytes é menos de meio kbyte!

Problemas da Alocação Contínua para registros

- Mas e com a manipulação de registros?
  - armazenando apenas 4 campos temos:
    - 150 bytes do nome
    - 4 bytes de cada inteiro (8 bytes)
    - 4 bytes de cada float (8 bytes)
    - **total de 166 bytes para apenas 1 registro com 4 campos!**
    - **Imagine o tamanho de um vetor de alunos de uma academia de rede com inúmeros campos para cada aluno?**

Problemas da Alocação Contínua para registros

- Suponhamos 500 mil alunos:
  - e um registro de 500 bytes por aluno
  - 500000*500 = 250 milhões de bytes
    - ~512 mil KBytes
    - ~500 MBytes (~meio GB)
      - **Impraticável reservar esse espaço em alocação contínua!**

Problemas da Alocação Contínua para registros

- Precisamos de uma estrutura que cresce e decresce de maneira dinâmica e unitária, aproveitando os espaços da memória de forma não contínua
  - conceito de **listas encadeadas**
    - Tema da nossa próxima aula!