Introdução à Lógica de Programação

Prof. Lucas Amparo Barbosa

Semestre letivo 2020.2

O QUE SÃO PONTEIROS?

- São variáveis que armazenam um endereço de memória
  - Todos os tipos de dados podem ser ponteiros;
  - Também podem apontar para “nada” (ponteiro vazio).

- Permite uma maior flexibilidade ao reutilizar um dado
  - Se várias partes do código tem ponteiros que apontam para o mesmo espaço de memória, quando atualizar um, atualiza todo mundo.

- Grandes poderes, grandes responsabilidades
  - A flexibilidade que os ponteiros permitem também são a causa de grande parte dos erros de implementação, mesmo para programadores experientes.

COMO DECLARAR UM PONTEIRO?

- O operador * na declaração do tipo define um ponteiro.

```
int * ponteiro_int;
float * ponteiro_float;
double * ponteiro_double;
string * ponteiro_string;
```

COMO UTILIZAR UM PONTEIRO DIRETAMENTE?

- Para acessar o valor de um ponteiro, também utilizamos o *

```
#include <iostream>

using namespace std;

int main() {
  int * numero;
  // inserindo um valor
  *numero = 20;

  // apresentando o valor salvo
  cout << *numero << endl;
}
```

COMO UTILIZAR UM PONTEIRO INDIRETAMENTE?

- Podemos armazenar no ponteiro o endereço de **outra variável**. Para isso, utilizamos &.

```
#include <iostream>

using namespace std;

int main() {
  int *numero, valor;
  valor = 20;

  // Informando que o ponteiro aponta para a variável
  numero = &valor;

  // apresentando o valor salvo
  cout << *numero << endl;

  // atualizando valor
  valor = 10;

  // apresentando o valor salvo
  cout << *numero << endl;
}
```

PONTEIROS E ALOCAÇÃO DE MEMÓRIA

- Podemos utilizar o conceito dos ponteiros para alocar memória apenas quando necessária

```
#include <iostream>

using namespace std;

int main() {
  int *numero;

  // Aloco a memória com new
  numero = new int;
  *numero = 10;

  cout << *numero << endl;

  // Libero a memória após utilizar com delete
  delete numero;
}
```

PRÁTICA 1: MANIPULAÇÃO DE PONTEIROS

Faça um programa que receba dois inteiros.
Utilize um ponteiro para armazenar o endereço da variável antes da entrada.
Utilize o ponteiro para apresentar os valores.

PONTEIROS E VETORES

- Seguindo a lógica da alocação de memória, podemos utilizar o conceito para reservar vários espaços de memória. Isso é um **vetor**.

```
#include <iostream>

using namespace std;

int main() {
  int *numero;

  // Aloco a memória com new
  numero = new int[20];
  numero[0] = 10;
  numero[10] = 20;

  cout << numero[0] << endl;
  cout << numero[10] << endl;

  // Libero a memória após utilizar com delete
  delete numero;
}
```

PONTEIROS E VETORES

- Veja que o exemplo anterior não utilizamos o * para acessar, e sim a indexação padrão de vetores. Ainda podemos utilizar o * e isso pode ser muito útil.

```
#include <iostream>

using namespace std;

int main() {
  int *numero;

  // Aloco a memória com new
  numero = new int[20];
  numero[0] = 10;
  numero[10] = 20;

  // Primeiro elemento
  cout << *numero << endl;
  // 10 elementos após o primeiro
  cout << *(numero + 10) << endl;

  // Libero a memória após utilizar com delete
  delete numero;
}
```

PONTEIROS E VETORES

- E se eu misturar os dois?

```
#include <iostream>

using namespace std;

int main() {
  int *numero;

  // Aloco a memória com new
  numero = new int[20];

  for(int i = 0; i < 20; i++) {
    *(numero + i) = i*i;
  }

  for(int i = 0; i < 20; i++) {
    cout << numero[i] << endl;
  }

  delete numero;
}
```

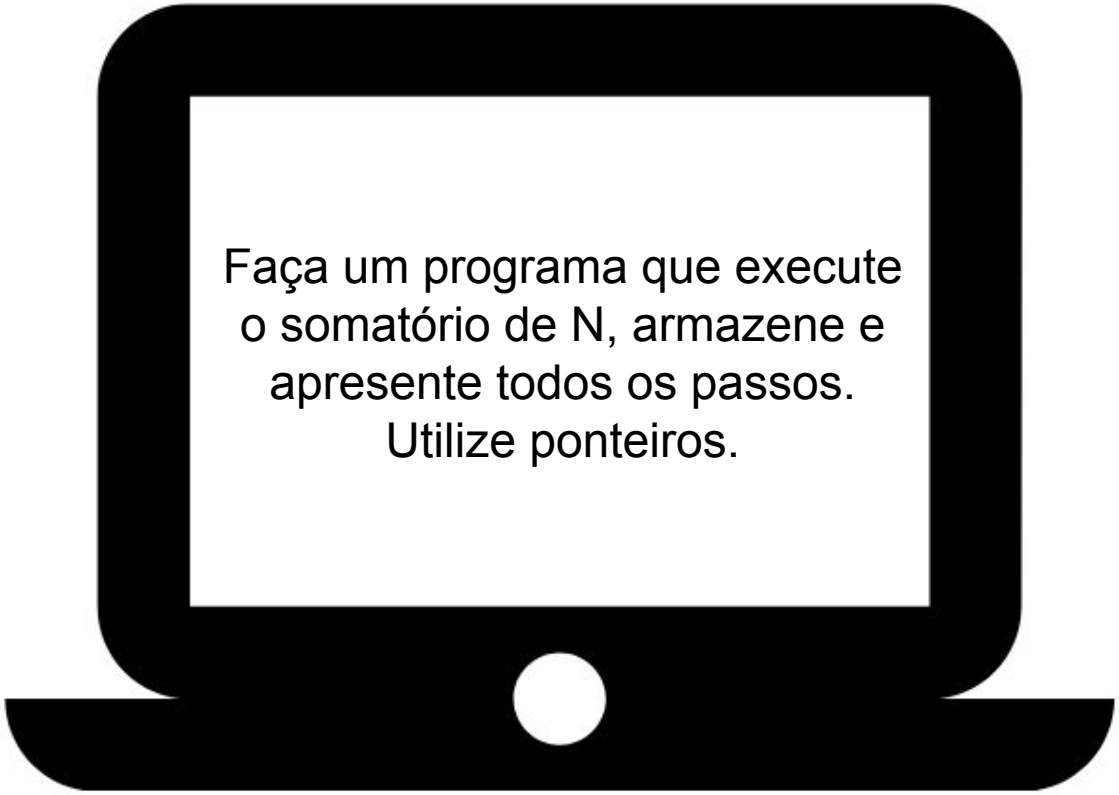
PONTEIROS E MEMÓRIA

- E se eu não alocar memória, funciona?
  - Vai acontecer um dos maiores erros para quem programa em C++: Segmentation Fault;
  - No URI, vai aparecer como “Runtime Error”.

- O que significa?
  - Significa que você está tentando acessar uma memória que não deveria, seja porque você não reservou memória o suficiente, seja porque está tentando acessar memória de outro processo.

- Por causa da natureza do ponteiro, o C++ permite que você compile códigos mesmo com acesso de memória equivocado.
  - Devemos sempre tomar cuidado redobrado ao utilizar ponteiros;
  - Se der para evitar, evite. Mas as vezes não tem para onde correr.

PRÁTICA 2: SOMATÓRIO



PONTEIROS E FUNÇÕES

- Podemos utilizar ponteiros nos parâmetros das funções;

- Melhoramos o desempenho do consumo de memória
  - Sem utilizar ponteiros, o programa faz uma cópia do conteúdo da variável para outra ao passar para a função;

- Podemos retornar mais de um valor no final da função
  - Se o endereço de memória é o mesmo dentro e fora da função, basta continuar utilizando a variável depois de processar

PONTEIROS E FUNÇÕES

- Melhoramos o desempenho do consumo de memória
  - Sem utilizar ponteiros, o programa faz uma cópia do conteúdo da variável para outra ao passar para a função;

```
#include <iostream>

using namespace std;

// Solicitando o endereço de memória
// da variável como parâmetro
void quadrado(int &n) {
    n = n * n;
}

int main() {
    int n = 10;

    quadrado(n);

    cout << n << endl;
}
```

PONTEIROS E FUNÇÕES

- Podemos retornar mais de um valor no final da função
  - Se o endereço de memória é o mesmo dentro e fora da função, basta continuar utilizando a variável depois de processar

```
#include <iostream>

using namespace std;

int divisao(int a, int b, int &r) {
    r = a % b;
    return a / b;
}

int main() {
    int a = 10, b = 3, r;

    int d = divisao(a, b, r);

    cout << a << "/" << b << endl;
    cout << "Resultado: " << d << endl;
    cout << "Resto: " << r << endl;
}
```

PRÁTICA 3: FATORAÇÃO