Programação Orientada a Objetos

Evolução do C para o C++: Alocação Dinâmica, Ponteiros, Arrays e Referências

1. Ponteiros em C e C++

1.1. O que é um Ponteiro?

Um **ponteiro** é uma variável que armazena o **endereço de memória** de outra variável. Ele permite acessar e modificar o valor da variável apontada.

1.2. Ponteiros em C

Declaração de Ponteiro:

```
int *ptr; // Ponteiro para inteiro
```

Exemplo prático em C:

```c
#include <stdio.h>

int main() {
    int valor = 42;
    int *ptr = &valor; // ptr recebe o endereço de valor

    printf("Endereco de valor: %p\n", (void*)&valor);
    printf("Endereco armazenado em ptr: %p\n", (void*)ptr);
    printf("Valor apontado por ptr: %d\n", *ptr);

    *ptr = 50; // Altera o valor da variável "valor" usando o ponteiro
    printf("Novo valor de valor: %d\n", valor);

    return 0;
}
```

Saída esperada:

```
Endereco de valor: 0x7ffeef4d0e4c
Endereco armazenado em ptr: 0x7ffeef4d0e4c
Valor apontado por ptr: 42
Novo valor de valor: 50
```

1.3. Ponteiros em C++

O conceito de ponteiro em C++ é idêntico ao C, mas o C++ oferece **referências** (discutido mais adiante) que facilitam o trabalho.

Exemplo prático em C++:

```cpp
#include <iostream>
using namespace std;

int main() {
    int valor = 42;
    int *ptr = &valor;

    cout << "Endereco de valor: " << &valor << endl;
    cout << "Endereco armazenado em ptr: " << ptr << endl;
    cout << "Valor apontado por ptr: " << *ptr << endl;

    *ptr = 50;
    cout << "Novo valor de valor: " << valor << endl;

    return 0;
}
```

---

2. Alocação Dinâmica de Memória

2.1. Alocação Dinâmica em C

Em C, usamos **malloc**, **calloc** e **free** para manipular memória dinamicamente.

Exemplo prático usando estrutura:

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int id;
    char nome[50];
} Pessoa;

int main() {
    Pessoa *pessoa = (Pessoa *)malloc(sizeof(Pessoa)); // Aloca memória para uma estrutura Pessoa
    if (pessoa == NULL) {
        printf("Memória não alocada\n");
        return 1;
    }

    pessoa->id = 1;
    snprintf(pessoa->nome, sizeof(pessoa->nome), "João");
    printf("ID: %d, Nome: %s\n", pessoa->id, pessoa->nome);
```

```
    free(pessoa); // Libera a memória
    return 0;
}
```

2.2. Alocação Dinâmica em C++

Em C++, usamos **new** e **delete** em vez de **malloc** e **free**.

Exemplo prático para instância de classe:

```
#include <iostream>
using namespace std;

class Pessoa {
public:
    int id;
    string nome;
};

int main() {
    Pessoa *pessoa = new Pessoa; // Aloca memória para uma instância da classe
Pessoa
    pessoa->id = 1;
    pessoa->nome = "João";

    cout << "ID: " << pessoa->id << ", Nome: " << pessoa->nome << endl;

    delete pessoa; // Libera a memória alocada
    return 0;
}
```

Exemplo prático para array de classes:

```
#include <iostream>
using namespace std;

class Pessoa {
public:
    int id;
    string nome;
};

int main() {
    Pessoa *pessoas = new Pessoa[3]; // Aloca memória para um array de 3 objetos
Pessoa

    for (int i = 0; i < 3; i++) {
        pessoas[i].id = i + 1;
```

```
        pessoas[i].nome = "Pessoa " + to_string(i + 1);
    }

    for (int i = 0; i < 3; i++) {
        cout << "ID: " << pessoas[i].id << ", Nome: " << pessoas[i].nome << endl;
    }

    delete[] pessoas; // Libera a memória alocada
    return 0;
}
```

Comparativo:

- **malloc/free** (C) vs **new/delete** (C++).
- O **new** não precisa de cast.
- O **delete** é usado para liberar uma única instância.
- O **delete[]** é usado para liberar arrays dinâmicos.

2.3. Arrays Dinâmicos em C++

A alocação de arrays dinâmicos em C++ é feita com o operador **new[]**. Cada elemento do array pode ser acessado por meio de notação de índice. Para liberar a memória alocada para o array, é necessário usar o operador **delete[]**.

Sintaxe geral:

```
int *array = new int[tamanho]; // Alocação de um array de inteiros
// ... uso do array
delete[] array; // Liberação da memória
```

Exemplo prático:

```
#include <iostream>
using namespace std;

int main() {
    int tamanho = 5;
    int *array = new int[tamanho]; // Aloca memória para um array de 5 inteiros

    for (int i = 0; i < tamanho; i++) {
        array[i] = i * 10; // Preenche o array com valores
    }

    for (int i = 0; i < tamanho; i++) {
        cout << "array[" << i << "] = " << array[i] << endl; // Exibe o conteúdo
do array
    }
```

```
    delete[] array; // Libera a memória alocada
    return 0;
}
```

2.4. Arrays de Objetos Dinâmicos em C++

Quando se trabalha com classes e objetos, o operador **new[]** permite alocar um array de objetos. Para liberar a memória de cada objeto, utilizamos o operador **delete[]**.

Exemplo prático:

```
#include <iostream>
using namespace std;

class Pessoa {
public:
    int id;
    string nome;
};

int main() {
    Pessoa *pessoas = new Pessoa[3]; // Aloca memória para um array de 3 objetos
Pessoa

    for (int i = 0; i < 3; i++) {
        pessoas[i].id = i + 1;
        pessoas[i].nome = "Pessoa " + to_string(i + 1);
    }

    for (int i = 0; i < 3; i++) {
        cout << "ID: " << pessoas[i].id << ", Nome: " << pessoas[i].nome << endl;
    }

    delete[] pessoas; // Libera a memória alocada
    return 0;
}
```

Importante:

- **new** cria uma única instância de um objeto ou variável.
- **new[]** aloca memória para um array de objetos.
- **delete** libera a memória de uma única instância.
- **delete[]** libera a memória de um array de objetos.

Comparativo:

- **malloc/free** (C) vs **new/delete** (C++).
- O **new** não precisa de cast.

- O **delete** é usado para liberar uma única instância.
- O **delete[]** é usado para liberar arrays dinâmicos.

---

3. Referências em C++

3.1. O que é uma Referência?

Uma **referência** em C++ é um **apelido** (ou "alias") para uma variável existente. Diferente de um ponteiro, uma referência não possui um endereço próprio, mas atua como se fosse o próprio objeto ao qual se refere.

Características importantes:

- **Deve ser inicializada no momento da declaração.**
- **Não pode ser nula.**
- **Não pode ser alterada para se referir a outro objeto.**

Sintaxe de Declaração:

```
int valor = 10;
int &ref = valor; // 'ref' é uma referência para 'valor'
```

3.2. Diferença entre Referência e Ponteiro

Características	Referência	Ponteiro
Necessidade de inicialização	Sim	Não
Pode ser nulo?	Não	Sim
Redirecionamento	Não	Sim (pode apontar para outro endereço)
Sintaxe de acesso	Direta (`ref`)	Indireta (`*ptr`)

3.3. Uso de Referências

3.3.1. Referência como Argumento de Função

O uso de referências como argumentos de função permite a **passagem por referência**, evitando a cópia de valores e permitindo a modificação direta do valor original.

Exemplo:

```
#include <iostream>
using namespace std;

void dobrarValor(int &num) {
    num *= 2;
```

```
}

int main() {
    int valor = 10;
    dobrarValor(valor);
    cout << "Valor dobrado: " << valor << endl;
    return 0;
}
```

Saída esperada:

```
Valor dobrado: 20
```

3.3.2. Referência como Retorno de Função

Uma função pode retornar uma **referência** para permitir que o chamador altere o valor original.

Exemplo:

```
#include <iostream>
using namespace std;

int& maior(int &a, int &b) {
    return (a > b) ? a : b;
}

int main() {
    int x = 10, y = 20;
    maior(x, y) = 100; // Modifica o maior valor (y)
    cout << "x: " << x << ", y: " << y << endl;
    return 0;
}
```

Saída esperada:

```
x: 10, y: 100
```

3.3.3. Referências Constantes

Referências constantes garantem que o valor referenciado não será alterado.

Exemplo:

```cpp
#include <iostream>
using namespace std;

void mostrarValor(const int &num) {
    cout << "Valor: " << num << endl;
}

int main() {
    int valor = 42;
    mostrarValor(valor);
    return 0;
}
```

Saída esperada:

```
Valor: 42
```

3.4. Referências em Arrays e Loops

Referências são úteis para percorrer arrays de forma eficiente.

Exemplo com loop `for` de referência:

```cpp
#include <iostream>
using namespace std;

int main() {
    int array[] = {10, 20, 30, 40, 50};

    for (int &elemento : array) {
        elemento *= 2; // Dobra o valor de cada elemento
    }

    for (int elemento : array) {
        cout << elemento << " ";
    }
    return 0;
}
```

Saída esperada:

```
20 40 60 80 100
```

3.5. Conclusão sobre Referências

As **referências** tornam o código mais claro e eficiente, especialmente em:

- **Passagem de parâmetros por referência** (evitando cópias desnecessárias).
- **Retorno de funções** (permitindo modificar variáveis externas).
- **Laços de repetição** (`for-each`), permitindo acessar e modificar diretamente os elementos do array.

Comparativo com Ponteiros:

- Ponteiros podem ser reatribuídos, referências não.
- Referências possuem sintaxe mais clara e evitam erros de desreferenciamento.

Compreender e utilizar as referências em C++ é essencial para um código mais legível, eficiente e seguro.

Exercícios

Exercício 1

Objetivo: Implementar um programa que permita ao aluno praticar a alocação dinâmica de memória, o uso de ponteiros, arrays e referências. O aluno deve ser capaz de manipular essas estruturas de forma integrada, tanto em C quanto em C++.

1. **Parte 1 (C):**
    - Escreva um programa que:
        - Peça ao usuário para informar o tamanho de um array de inteiros.
        - Use `malloc` para alocar dinamicamente a memória para o array.
        - Preencha o array com os quadrados dos números de 0 a (tamanho-1).
        - Imprima os valores do array.
        - Libere a memória alocada com `free`.

Exemplo de entrada e saída (C):

```
Informe o tamanho do array: 5
array[0] = 0
array[1] = 1
array[2] = 4
array[3] = 9
array[4] = 16
```

---

2. **Parte 2 (C++):**
    - Converta o programa de C para C++.
    - Use `new` e `delete` para substituir `malloc` e `free`.
    - Implemente uma função que receba uma referência a uma variável e a dobre.
    - Após imprimir os valores do array, use a função para dobrar o valor do último elemento do array.
    - Imprima o array novamente.

Exemplo de entrada e saída (C++):

```
Informe o tamanho do array: 5
array[0] = 0
array[1] = 1
array[2] = 4
array[3] = 9
array[4] = 16
Dobrando o último valor...
array[4] = 32
```

Exercício 2

Objetivo: Implementar um programa em **C** e **C++** que utilize **structs** (em C) e **classes** (em C++) para manipular informações de **alunos**. O aluno deverá aplicar conceitos de **alocação dinâmica de memória**, **ponteiros**, **arrays dinâmicos** e **referências** para criar, manipular e liberar as estruturas de dados.

1. **Parte 1 (C):**
    - Implemente uma **struct Aluno** que contém:
        - **ID** (inteiro)
        - **Nome** (string com 50 caracteres)
        - **Nota** (float)
    - O programa deve:
        - Solicitar ao usuário a quantidade de alunos.
        - Alocar dinamicamente memória para armazenar esses alunos.
        - Permitir que o usuário insira os dados de cada aluno (ID, Nome e Nota).
        - Exibir a lista de alunos cadastrados.
        - Liberar a memória alocada.

---

2. **Parte 2 (C++):**
    - Converta o programa para C++ utilizando uma **classe Aluno** com os mesmos atributos (**ID**, **Nome**, **Nota**).
    - Implemente métodos da classe para:
        - **Definir os atributos** (setID, setNome, setNota).
        - **Exibir os dados** (método `exibirDados`).
    - O programa deve:
        - Solicitar a quantidade de alunos.
        - Alocar dinamicamente o array de objetos **Aluno**.
        - Usar **métodos de classe** para definir os dados de cada aluno.
        - Exibir a lista de alunos.
        - Usar **delete[]** para liberar a memória.

Orientações

1. **Compare as diferenças entre a struct e a classe**:
    - Em **C**, usamos ponteiros e o operador `->` para acessar os campos do aluno.
    - Em **C++**, a classe encapsula os métodos e atributos, e podemos usar métodos (`setID`, `setNome`, `setNota`, `exibirDados`) para manipular os dados.
2. **Alocação e liberação de memória**:
    - **C**: Alocação com `malloc` e liberação com `free`.
    - **C++**: Alocação com `new[]` e liberação com `delete[]`.
3. **Diferença de entrada de dados**:
    - **C**: Usa `scanf` para capturar strings e precisa de cuidados com buffer (`%s`).
    - **C++**: Usa `cin` e `getline` para capturar strings, sendo mais seguro e intuitivo.

Questões para Reflexão

1. Qual é a principal diferença entre uma **struct** e uma **classe**?
2. Por que é importante usar o operador **delete[]** em vez de **delete** ao liberar arrays?
3. O que acontece se esquecermos de liberar a memória alocada? Como isso afeta o programa?
